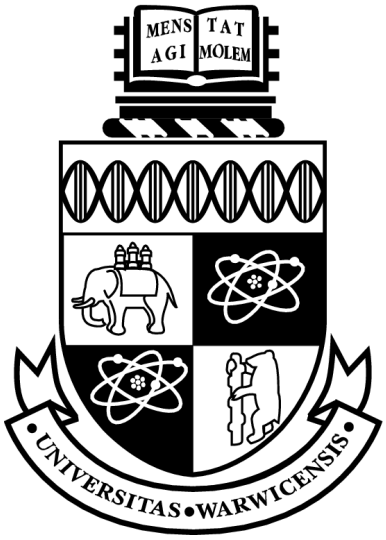

Supporting the Migration from Construal to Program: Rethinking Software Development

by

Nicolas William Pope

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

August 2011

THE UNIVERSITY OF WARWICK

Contents

List of Tables

List of Figures

Listings

Acknowledgments

More than any other I am indebted to my supervisor Meurig Beynon for his exceptional effort in guiding me through to completion and for the personal support he has given during the difficult times. Thanks must also go to Steve Russ who has given invaluable advice and suggestions through his efforts to get to grips with my work. Many others have been directly involved with the work. Special thanks must go to Sam Gynn who helped develop the game library and believed in my ideas. Tim Monks, David Evans, William Dangerfield, Alan Hazeldon and all those who have been burdened with using my work for their studies or in the Warwick Game Design society. In the early stages I am grateful for the input of Antony Harfield and Tim Heron in formulating the basis for this thesis. I am also grateful for the technical support given by Roger Packwood and the other staff in the department, as well as Russel Boyatt. Those friends who have supported me the most, for which I am grateful, deserve a mention here. Anne Kunz, Matthew Metcalfe, Colin "Sparky" McKenzie, Gareth Jones, Jenny Cao, Jenny Jones and more recently Ilaria Spalla. Thanks to all of you and those of Warwick Mountains who have kept me at least slightly sane. Finally I must thank Mum and Dad for all they have done over the past decades to make this possible.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself except where otherwise stated. All quoted text remains the copyright of the original attributed author. I have taken the liberty of correcting typographical errors, spelling and grammar where necessary. All trademarks are acknowledged. The walk analogy used in §2.2.1 and a general outline of the Cadence tool has been published in [Pope and Beynon, 2010b]. No other material in this thesis has been published, but some joint work which motivates and further explores the work contained within this thesis has been published in [Beynon, 2011] and the research report [Beynon and Pope, 2011].

Abstract

Creative software design, where there is no theory, no pre-computer precedent, no set of requirements or even necessarily an objective, challenges all existing software development methods. There can be no assumption that end-users know what they want. Each and every situation is unique, unpredictable and due to feedback is continually changing. Fixed solutions developed by non-domain experts are all but impossible in more unconventional systems, and increasingly there may not be domain experts at all. Allowing individuals or groups of non-professionals to program is one approach (End-User Development). However, programming requires a degree of formality, design and specification that cannot co-exist with the most informal pre-theoretical applications which need to be developed by exploratory experimentation to help with problem-solving and sense-making. Instead of programming a finished application from the beginning, there is a need to develop personal, provisional and subjective models and evolve these into public, objective and assured applications. Developing these models "on-line" through interactive experimentation is essential and it is the objective of Empirical Modelling (EM) research to enable the modelling of sense-making artefacts called *construals*.

Whilst existing EM tools are able to support construals there is a need to see how a smooth transition from construals to applications can be made. Such a migration is not one-way as the resulting applications need to remain plastic. The aim of this thesis is to explore and develop ways of enhancing EM principles and tools to better support such migrations from construals to programs.

By first identifying key characteristics of construals and associated principles and techniques, along with a critique of the existing EM tool, a new kind of environment for plastic software development is proposed. A major contribution of this thesis is the development of such a prototype environment which is illustrated using a collection of artefacts developed within it. From the prototype, called Cadence, an informal and a formal idealised account was elicited to provide a framework for this kind of development activity. The ideas explored in the thesis have the potential to impact upon the operating systems community and the everyday computer user in radical ways if taken forward. The thesis demonstrates that applications can be developed from construals without a translation step, keeping the resulting applications plastic.

Abbreviations

CINE	Cadence in EDEN
DASM	DOSTE Assembly
DMT	Dependency Modelling Tool
DOSTE	Definitive Object State Transition Engine
EDEN	Evaluator of DEfinitive Notations
EM	Empirical Modelling
EUD	End-User Development
HDR	High Dynamic Range
IDE	Interactive Development Environment
OD-net	Observable Dependency Network
ODA	Observable Dependency Agency
OID	Object Identifier
PSE	Plastic Software Environment
AOP	Agent Oriented Parser
ADM	Abstract Definitive Machine

HCI Human Computer Interaction

WPF Windows Presentation Foundation

Chapter 1

Introduction

This thesis identifies and develops specific ways of improving Empirical Modelling tools and concepts as a *plastic software environment* to better support *plastic applications*. The terms *plastic software environment* (PSE) and *plastic applications* (plastic apps) have been introduced by this thesis to classify a certain kind of software and software environment, the definitions of which are given in this chapter in order to place the rest of the thesis in context. Empirical Modelling is an existing area of research that provides a good foundation for the support of plastic apps and will be briefly introduced here with more detail given in chapter 2. The remainder of this chapter will then discuss more specific thesis aims and questions and give an outline of each chapter.

1.1 Plastic Applications

The notion of *plastic applications* is for the most part an extrapolation from what is currently possible, and relates to the vision of making *software soft* [Fischer, 2009]. It is a vision where an artefact can be sculpted in a creative fashion from the ground up by end-users, where the artefact gradually evolves and solidifies into a useful, non-trivial, sophisticated application. In the beginning a plastic app could be some informal experimental model developed by a user who can then use this to learn about their specific

problem through continued experimentation. At any point, even after it has solidified to a usable application, the application can be adapted and moulded into something new. This subsequent adaptation is not so plausible in traditional programs that have been developed and optimised specifically for a machine. The result of enabling plasticity is a concrete contextualised application as opposed to a generic off-the-shelf product. An additional part of this vision would be the ability to link many different plastic apps together in unforeseen ways to produce an entirely plastic computing environment, perhaps even a plastic operating system[1].

The concept of plastic apps comes from a collection of research areas, including end-user application development (EUD) [Lieberman et al., 2006], formerly end-user programming (EUP). Nardi gives a clear introduction in her book entitled "A Small Matter of Programming" [Nardi, 1993] as to why end-user programming is vital. More recent work by others discusses how, in specific cases such as with spreadsheets, it has proven remarkably successful [Burnett, 2009]. One such argument given by Nardi is that end users "have the detailed task knowledge necessary for creating the knowledge-rich applications they want" (p.xi) and that "the process of transferring domain knowledge to a programmer... is inefficient" (p.123). Perhaps, however, we are dealing with end-users who do not have a detailed task knowledge but who might be attempting to understand a particular solution to a problem whilst they are developing the application? In which case EUD is also a learning experience [Repenning and Ioannidou, 2006]. A claim made by Nardi that I strongly agree with, and which is still valid today, is that "we have only scratched the surface of what would be possible if end users could freely program their own applications" [Nardi, 1993, 3].

EUD has been defined as "*a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artefact*" [Lieberman et al., 2006]. It has been claimed by some that EUP and perhaps EUD has succeeded [Burnett, 2009]

[1]Operating systems may already be considered as plastic, but not to the degree being proposed in this thesis.

as we now have spreadsheets, web authoring tools, graphical languages and various educational tools. To a degree this is true - there are now over 50 million end-user programmers who adapt or create software in some form. However, is the software they are creating or adapting really that plastic? Certainly as an everyday user I am not often able to adapt software to my personal needs except in the most specific and pre-defined ways such as with e-mail filters or user interface customisation, or by learning how to program and write custom extensions. The problem seems to be that EUD systems in practice are typically domain restricted or only support a tiny amount of end-user adaptation of existing environments [Repenning and Ioannidou, 2006]. What is needed then is an environment which supports the extreme notion of *plastic applications* which is not, as a system, domain specific and not restricted to specific developer chosen adaptations. EUD needs to be broadened beyond adaptation of existing environments. There have been attempts to do this in the EUD community [Burnett et al., 2001][Repenning et al., 2000] but such research has not gained widespread recognition in the software industry as a whole [Fischer, 2009]. More will be said on these tools and others in chapter 2.

Other researchers in the Human Computer Interaction (HCI) and software development communities have adopted terms such as "software plasticity" to describe the degree to which an interactive piece of software can adapt to changing contexts of use [Calvary et al., 2004][Morris, 2005][Sendn et al., 2005], although this is often in a developer focused context rather than that of an end-user. There is also now widespread use of technologies whose purpose is to increase flexibility to better deal with change, including Service-Oriented Architectures [Erl, 2005], Aspect-Oriented Programming [Kiczales et al., 1997] and Agile methodologies [Dingsyr et al., 2010], all of which have the potential to radically improve end-user application development but currently, for the most part, remain in the domain of professional software developers. There are a lot of technologies out there which could help enable plastic applications but that have yet to be used in this way.

One additional area of research which provides a background for the plastic ap-

plications concept is that of Empirical Modelling (EM) at the University of Warwick [EM Website][2]. The Empirical Modelling group have been exploring informal modelling tools and principles which enable users to develop models without needing to be expert programmers, or experts in the model's domain. A key aim of Empirical Modelling is to allow users to develop models informally without specification or design so that they may gain understanding of a problem they know little about prior to the modelling activity [Milner, 1986][Beynon and Russ, 2006]. EM is a conceptual framework for end-user development but rather than looking at ways of adapting traditional applications it has taken the more radical step of rethinking programming [Beynon et al., 2006a] and software from the ground up. The ideas and tools behind EM provide a great deal of insight into possible ways of implementing the EUD vision and the vision of supporting plastic applications. There are already several example EM models [EM Website], developed using existing tools [Ward, 2004, p.194], which illustrate the plastic app concept, including one model for timetabling which has been used as an application by staff at the University of Warwick [Beynon et al., 2000b]. The EM concepts and tools, however, have not yet been able to support the development of fully-fledged applications from models, in part because of the EM research objectives not being aligned with those of EUD but focused more on personal and individual modelling activities, and in part due to inadequacies of the tools.

1.2 Plastic Software Environments

A *plastic software environment* (PSE) is defined here to be, as the name suggests, an environment that hosts and enables the continuous development of one or more plastic applications. Such an environment may just be a form of modelling tool or virtual machine but could be an entire operating system hosting many different but connected plastic applications. The principles and concepts behind a PSE will be discussed further in section 2.1.1 which identifies key EUD principles and also towards the end of this

[2] An excellent introduction to EM can also be found in [Harfield, 2008, p.31]

work in 6.1. Some PSE characteristics, however, include:

- supporting the reversable refinement of an application.
- using a single conceptual model allowing gentle-slope development.
- not being brittle in the presence of the unexpected.
- being live and interactive for immediate feedback.
- not requiring initially undesirable precision and formality.

Whilst there are many examples of environments and applications that support end-user development to some degree, for example GIMP, Kate and Google-Mail (with their supporting technologies), there are few that would be considered a PSE. Examples of a PSE are hard to find but include spreadsheet environments, Forms/3 [Burnett et al., 2001] and AgentSheets [Repenning et al., 2000] along with the EM tool Eden [Ward, 2004, p.194]. With regards to spreadsheets, plastic applications would be models that have been sufficiently developed to become useful applications, perhaps for financial modelling or a teacher developed model for a school sports day. In these examples the model may well be used as an application but is always open to modification to adapt, extend or fix at any time.

Apart from perhaps spreadsheets, plastic software environments are far from ubiquitous. With the enormous success of the spreadsheet it seems most surprising that EUD environments have not become increasingly powerful and available even though, in general, software is now moving towards EUD. Perhaps the reason is the lack of research into the principles and conceptual frameworks for them?[3] The key players should be the operating systems developers and communities since a plastic operating system[4] would be a "dream come true".

[3] Of course it is also necessary for such software to be marketed.

[4] A plastic OS would be one with extreme end-user flexibility, beyond configuration files, scripts and utilities to a comprehensive modelling environment. There is some connection to SmallTalk and similar all inclusive environments.

1.3 A Lack of Plasticity

In order to achieve the level of flexibility (or plasticity) desired for plastic apps there is a need to look more closely at why existing systems are not capable of providing it, ignoring for the moment the environments developed by the EUD community (discussed in chapter 2). To aid this discussion let us use an analogy of developing a recipe for making a cup of tea. It is relatively easy to provide a generic recipe for making tea that is along the lines of: fill kettle with water, turn kettle on, get tea bags and mugs, pour boiling water from kettle into mugs, add tea bag to mug, remove tea bag, add milk and so on. Such a *program* could be parameterised by the developers to allow the user to control, for example, how long to leave a tea bag in the water, how much milk to use, how many mugs, how much water and so on. What if, however, the person making the tea was in an unfamiliar house and did not know where to find the tea or there is no kettle and so a sauce pan is needed instead?

One of the problems is that programs are based upon brittle assumptions. David King, in his thesis on "Parting Software and Program Design" [King, 2005], states that "the assumptions underlying programs are always brittle" (p.10) because "program descriptions must ... be both closed and complete" (p.14). Whilst "brittle assumptions [are] strong when true, but useless when false" (p.10) it is unfortunate that 'almost every 'interesting' system is incomplete" (p.14) meaning that these assumptions are likely to fail to hold. From the analogy, the *program* for making tea makes assumptions about knowing where to find tea bags and about the existence of a kettle. It is conceivable that the developers of the program foresaw this and provided the option of using a saucepan, but what if they did not? At this point, in a traditional application, the developers would need to alter the requirements specification, design and implementation of the program to allow for saucepans.

The intertwining of implementation and specification has been recognised for decades [Swartout and Balzer, 1982]. Lehman proposed the concept of software evolution back in 1980 [Lehman, 1980] (although the term was used prior to this), along with

a set of associated laws of software evolution which state that change is inevitable in certain kinds of applications and results in the need for an iterative approach to software development. One of the more significant laws, in my opinion, is that of feedback where the introduction of a piece of software will, by its very existence, change the nature of the problem [Lehman, 1996]. The consequence of this feedback is that it becomes impossible to predict in advance the requirements and design of an application [Fischer, 2009]. Since Lehman first studied the concept of software evolution there has been a great deal of research into software evolution, including empirical studies and the development of tools and techniques for dealing with evolving programs. The software industry has also recognised the problem of changing requirements, which has led to the development of technologies such as the recent Service-Oriented Architectures approach, Aspect-Oriented Programming and other kinds of Object-Oriented Programming, along with Agile methodologies to rapidly implement these changes. All of these new technologies and methodologies are there to reduce the difficulty and increase the speed with which changes can be implemented by developers. All of these technologies and methodologies are there to alleviate the problems associated with having brittle assumptions, the result of needing to use traditional programs.

This is where end-user programming comes in. Allowing end-users to make these kinds of change instead of having to go back to the developers is an efficient way of dealing with change. However, all this does is shift the burden from the developers to the end-users so there needs to be a fairly radical change to the way software is developed if this is to work or for there to be any real improvement. Shifting the burden to end-users has led to considerable research into ways of making programming easier for these non-professional developers, such as programming-by-example [Lieberman, 2001], visual programming [Resnick et al., 2009] and domain-specific languages [Mernik et al., 2005]. The problems of brittle assumptions and lack of flexibility still remain with EUP, even if the tools are now easier to use. Getting to the heart of the problem with programming involves more than simply getting the end-user involved and requires a more radical

rethink about the nature of programming itself. The EUP community have realised this and it is partly why EUP became End-User Development. The notion of evolution and experimentation came in and the whole development process, not just the programming, came under scrutiny.

The notion of software as being engineered is often misconceived by traditional computer science [Jackson, 2005]. Software engineering focuses on theory and formality through specifications of requirements using mathematical abstractions. Whilst for some applications this is an appropriate strategy, for many existing applications and for many that have yet to be conceived it is wholly inappropriate to think that they could be engineered in this way. Engineering practice is about the design and construction of an artefact [Rogers, 1983] which transforms the physical world, rather than focussing on the machine world. As Jackson puts it, requirements and specifications "are all to be understood in terms of physical phenomena rather than in terms of purely mathematical abstractions ... [any] abstractions must be firmly grounded in observable physical reality" [Jackson, 2005]. As Lehman's laws indicate, change is inevitable and potentially rapid in the software world. With software we are also dealing with the unknown where there is sometimes no pre-computer precedent and where the consequences cannot be foreseen. Such applications cannot be formally engineered but must be explored and evolved by experiment to first gain understanding, more akin to the real practice of engineering. Despite this realisation by many (especially in the EUD community) the idea of formal design and specification remains entrenched.

David King provides some insight into this when he proposes to abandon the relationship between software design and program design in order to gain a fuller appreciation of the problem [King, 2005, p.85]. King's very next statement, however, is that "for nearly all design methods, this is a step too far" (p.85) because "for software design to have any value, we must have some ability to translate between software and program design" (p.60). Perhaps it is this idea that we need traditional programs and that they need to be designed which is the problem? If the design and development

stages were fully intertwined so that the idea of needing to develop programs separately as an additional step were to be removed then the problem of brittle assumptions would also be removed. Applications evolve out of models without any separate translation into a program, this is what the concept of plastic applications is about. A suggestion made by King is that instead of attempting to reduce complexity there need to be tools which better enable us to deal with that complexity. Perhaps we need plastic software environments which appropriately manage complexity whilst being end-user friendly and supporting the experimental evolution of a model from the ground up until it becomes a plastic application, an application that is not made brittle by its translation into a traditional optimised program.

1.4 Thesis Aims

The higher aim behind this work is to answer the question of *how to bring extreme plasticity to software in a way amenable to everyday users*. One of the difficulties in bringing this about is the lack of suitable plastic software environments in which such an activity of moulding software can take place. There are many applications that provide possible examples of a plastic or semi-plastic environment but most are domain-restricted. One of the most promising but underdeveloped approaches is that of Empirical Modelling (EM). In addition to a lack of tool support for plastic applications there is also a lack of principles and of a conceptual framework for such an activity. Empirical Modelling may also provide a basis for such a framework. So my research question is:

> How to adapt, in specific and selective ways, Empirical Modelling concepts and tools to enable a migration from informal models to programs by end-users as an attempt at supporting the creation of plastic applications?

There are numerous problems with existing Empirical Modelling tools as well as some issues with its principles which prevent it from scaling up to be considered

a fully-fledged plastic software environment to be used in real situations. This work explores these issues and proposes possible solutions in the form of a new tool and re-conceptualisation of the principles. An additional aim that is being kept in mind (but not a key focus) throughout the work is the possibility of developing a plastic operating system, which will influence the design of the new tool to some degree. Specific objectives include:

- Critiquing of Empirical Modelling (and to a lesser degree EUD) to identify specific areas of improvement and ways that these improvements can be achieved by the development of a new tool.
- Developing a new prototype tool for Empirical Modelling and EUD which attempts to implement solutions to the specific identified problems.
- Creating models and examples within the new prototype environment to demonstrate it as an EM EUD tool and to demonstrate the new features it provides.
- Analysing the work to identify key principles and concepts behind the new environment to develop a framework which can help support the notion of plastic applications.

1.5 Thesis Outline

The thesis has been organised into 8 chapters which are as follows:

Chapter 1 is this introductory chapter where the motivations and aims are discussed along with definitions for *plastic applications* and *plastic software environments*.

Chapter 2 goes through recent work of the EUD community by looking at principles, guidelines and tools which they have developed and which provide useful material for directing this work. Empirical Modelling is also introduced in more detail to explain its key principles and current tools. Other relevant technologies such as programming languages and other software industry techniques are introduced throughout the thesis.

Chapter 3 explores specific problems with existing approaches in achieving the kind of plasticity being sought after. Primarily the focus is on Empirical Modelling (EM) tools and concepts, where a critique is given, as these are already a close match to a plastic software environment. The problems with EM are then critiqued with reference to the solutions from industry and EUD that appeared in the background material of chapter 2. At the end of the chapter specific research questions are posed and a form of specification is given for a new prototype environment called Cadence.

Chapter 4 forms a central part of this thesis. It documents the development of Cadence, proposed in chapter 3. The architecture and interfaces for Cadence are given, along with some examples of how Cadence is to be used.

Chapter 5 discusses various example models that were developed within Cadence in detail. These models include games, presentation environments and biological models. The examples show the flexibility and other characteristics of the tool which are relevant to both the problems identified in chapter 3 as well as the broader objectives of the thesis. Each model has been used to illustrate how the Cadence tool has resolved specific issues.

Chapter 6 identifies key principles and concepts from the prototype to develop an idealisation of Cadence. These principles have come from the work in chapters 3,4 and 5 and provide a possible framework for plastic software environments generally. It is this chapter that contains the second major contribution of this thesis.

Chapter 7 looks at the impact of the Cadence tool and the principles of chapter 6 upon Empirical Modelling. Hybrid tools are discussed which attempt to resolve limitations in both Cadence and existing EM tools, as well as provide a clear means of comparison. Example models are also included that were developed by students of Empirical Modelling who used either Cadence or a hybrid for their projects and coursework.

Chapter 8 summarises the work of all previous chapters and evaluates it with respect to the broader objectives identified in this chapter and the specific problems and questions posed in chapter 3. Further work is identified here, along with the contributions and limitations of the work contained within this thesis. A brief statement on applications

and future directions is also given.

Chapter 2

Background

The previous chapter introduced two key areas that are the focus of this work: End-User Development and Empirical Modelling. As the aim is to develop new tools and principles it is important that the existing tools and principles of both areas are understood since they provide the foundation for this work. To that end this chapter will begin by giving a brief summary of key EUD principles, guidelines and relevant existing tools, followed by a similar summary of EM principles and tools.

2.1 End-User Development

Today the focus of the End-User Development community is on End-User Software Engineering (EUSE) [Ko et al., 2011][Burnett, 2009]. It is generally accepted by the community that to a large degree the issue of end-user programming has been adequately dealt with for the time being and now that millions of end-users are programming there are other issues to contend with. These issues relate to the specification, design and testing by end users of their software because one of the major concerns is that of the quality and reliability of end-user developed applications [Burnett, 2009]. However, it seems that this focus is on a minority of end-users who are developing sufficiently large applications in critical situations using EUP technologies. The vast majority of end-

users, especially those at home making personal use of a computer, are still struggling with EUD and hence the focus of this work being to broaden the scope of EUD beyond professionals and education.

As this work is looking at how to merge EUD concepts and Empirical Modelling concepts to produce a plastic environment, the grander issues of design and testing in larger projects are going to be put aside. The focus for this work will remain on the principles and guidelines of EUD and end-user programming, along with the technologies that exist which enable EUD, to see how these can be applied to EM or how EM already uses them.

2.1.1 Principles

There are a collection of key principles that can be found across the EUD literature which are important for enabling end-users to adapt and develop their own systems. The principles give a vague indication of a framework for EUD, although this remains rather loose and poorly formulated. One attempt at developing such a framework is that of *meta-design* by Fischer [Fischer, 2000][Fischer and Giaccardi, 2006][Fischer, 2009]. The most significant and relevant principles are given below.

- Gentle Slope: for an incremental increase in the complexity of an adaptation there should be a similarly incremental increase in the complexity of the mechanisms to achieve it [Dertouzos, 1997][Pane and Myers, 2006][Lieberman et al., 2006]. The main point to gain from this principle is that the user should not suddenly reach a point where they have to modify source code and recompile the application, this is a sudden leap in the complexity involved in modifying the program and is a steep learning curve that many people will be unwilling to overcome. The EUD approach to resolving this usually involves different levels of adaptation, starting with parameterisation for customising certain interface and application features, then allowing users to reorganise components and finally to allow for scripting and modules [Spahn et al., 2008]. Whilst these techniques do produce a more gentle

slope than traditional programming, they are far from a smooth slope and each requires different skills that need to be learned. The more adaptation mechanisms involved the more gentle and smooth the slope of complexity will be, but it is vital that the principles underlying all these mechanisms are related so that there can be an elegant transition which builds upon the previous [Spahn et al., 2008].

- Liveness: Also known as *live editing* [Smith et al., 1995] where changes can be made to a live system without needing to restart the application [Lieberman et al., 2006; Tanimoto, 1990]. This enables users to see and directly experience the consequences of any changes with immediate feedback [Burnett et al., 2001] and allows those changes to be of an incremental nature. It is one of the key characteristics of spreadsheets that has made them so successful [Nardi, 1993, p.88]. In order to achieve live editing a certain degree of robustness is required along with the ability to undo mistakes. It also goes hand-in-hand with the *gentle slope* principle since not allowing for live editing will certainly introduce barriers for the user.

- Directness: by reducing the conceptual distance between the problem domain and software environment the user can more easily identify and resolve their problems by making appropriate changes [Pane and Myers, 2006]. Directness comes from the HCI concept of direct manipulation which attempts to make user interfaces easier to use and is related also to the gulfs of execution and evaluation [Norman, 1998]. It has not been a consideration in the design of most programming languages so the programming world and problem world remain distanced which hinders the problem solving process [Pane and Myers, 2006; Green and Petre, 1996][1]. It is a vital principle in EUD and can take various forms including task-specific languages [Nardi, 1993, p.37] or concrete and directly manipulable objects [Smith et al., 1995].

[1] Programming languages are not usually designed with problem solving in mind.

Table 2.1: EUD Principles

1	Gentle Slope
2	Liveness
3	Directness
4	Evolutionary Development
5	Learning Experience

- Evolutionary Development: sometimes this is described as *design-during-use* or *design for change* [Dittrich et al., 2006][Fischer and Giaccardi, 2006]. Evolution is at the heart of EUD and is the driving factor behind it. Things are always changing, either because the environment is changing or because the focus is changing [Lehman, 1980][Fischer, 2009]. Not only is there change but often it is not possible to adequately identify all requirements to design a piece of software in advance [Swartout and Balzer, 1982], especially in EUD where in the most extreme cases there are no requirements since it becomes an experimental hobbyist activity [Ko et al., 2011]. As a consequence evolution must be supported in software applications and allowing end-users to customise and extend programs is an efficient way to achieve this [Nardi, 1993, p.3].

- Learning Experience: the whole process of EUD can be framed as a learning experience [Repenning and Ioannidou, 2006]. This can be in the sense of learning the tools and how to program, or learning about the specific domain of the application as development takes place. There has been a considerable focus in EUD on educational technologies that allow students to develop simulations or, through domain specific languages, develop their problem solving and programming skills.

2.1.2 Common Approaches

The technologies used for enabling EUD typically fall into one of the five categories which are listed below [Spahn et al., 2008]. It is significant that these approaches are either exceptionally simple but restricted to what the developer anticipated, or involve some form of simplified programming. Despite "gentle-slope" being a key principle it seems there is still a gap between parameterisation and programming. For the most part the simplified programming approaches are also domain-specific and there is yet another gap between them and more sophisticated scripting and programming languages. The bridge between interface customisation and classical programming is rather weak.

- Interface Customisation
- Application Parameterisation
- Programming-by-Demonstration
- Visual Programming
- Natural Language Programming

2.1.3 Existing Environments

Both within the EUD community and outside it there are many examples of adaptable applications making use of the principles and approaches identified above. These applications include GIMP[2], Kate, Firefox, Gmail and many more with most modern applications supporting some level of EUD. Whilst there have been interesting recent developments, such as Service-Oriented Architectures [Erl, 2005] (specifically web services) and the associated end-user developed web mash-ups, for this work of bringing EM into the EUD picture the focus will remain with the major and classic EUD environments and languages. Three of these environments have been identified as key players

[2]GIMP allows for extensive interface customisation, provides many parameters to control the application, allows users to write custom scripts in Scheme and supports user developed modules.

and are of specific interest due to some close connections with Empirical Modelling ideas: Forms/3 [Burnett et al., 2001], Self [Ungar and Smith, 1987] and Subtext [Edwards, 2005]. Other environments which have had some influence on the work include: AgentSheets [Repenning et al., 2000], HANDS [Pane and Myers, 2006], Croquet (now called Open Cobalt), Plan 9 [Pike et al., 1995], Singularity [Hunt and Larus, 2004] and EROS [Miller et al., 2003].

As a research language that is based upon the spreadsheet paradigm, Forms/3 attempts to remove some of the limitations encountered with other similar spreadsheet languages. The spreadsheet paradigm and associated languages are defined by Alan Kay's value rule [Kay, 1984] where a cell's value is defined solely by its formula. According to [Burnett et al., 2001] spreadsheets suffered from two limitations: first is the lack of support for different types and the second is the lack of abstraction mechanisms. It was the goal of Forms/3 to remove these limitations whilst remaining faithful to the value rule and also to the EUD and HCI principles already identified (gentle slope, liveness and directness).

In order to achieve its goals Forms/3 made two basic changes to the standard spreadsheet along with three major additions. The two basic changes are:

1. Removing the grid. Instead of forcing a grid layout onto cells it is possible to position the cells anywhere on the form (aka. worksheet). This enables additional flexibility in visualising results.

2. Giving cells names. Since there is no grid, cells can be given names to identify them.

The three major additions, which provide insight into ways of enhancing Empirical Modelling tools[3], are:

1. Graphical and user-defined types. Cell values may be a graphical type instead of being restricted to numbers and strings as is usually the case in spreadsheets. This

[3]Empirical Modelling tools are also based upon spreadsheet concepts so these enhancements are directly relevant.

allows for graphics without requiring features outside of the spreadsheet paradigm (breaking the *value rule*).

2. Dynamic Grids. The size of the grid may vary automatically and it is possible to give formula to a region without manual copy/paste actions.

3. Temporal streams of cell values. This allows for change over time and hence animation as well as "time travel".

User-defined types are supported in Forms/3 by the use of *type definition forms*[4]. A type is a collection of cells (or cell groups) which can internally reference each other. When a type is instantiated these internal cells may be connected to other cells by a formula reference. Whilst Forms/3 does provide these visual type definition forms and direct manipulation capabilities for creating types, it is all still describable as a textual cell formula and so still follows the *value rule* of a spreadsheet. By sticking with the use of forms, cells and formula for describing types the end-user is not burdened with new concepts such as classes. More recently the Forms/3 type system was extended to include support for inheritance, called *similarity inheritance* [Djang and Burnett, 1998][Djang et al., 2000].

The dynamic grids supported by Forms/3 are similar to traditional matrices and spreadsheet grids, only they are not statically determined. An important feature of these dynamic grids is the ability to specify a formula over a contiguous region (or the whole grid), which removes the formula replication task of the user but also allows for arbitrarily large grids with formulas as they are created in a lazy manner. These dynamic grids offer the same functionality as lists in functional languages since Forms/3 does support recursion. However, when used in combination with the time-varying properties of Forms/3 it has been found that both recursion and iteration are not required and that this improves directness and concreteness which helps with EUD [Burnett et al., 2001].

[4]Also called Visual Abstract Data Type (VADT) forms and are discussed in more depth in [Djang et al., 2000].

Cells in Forms/3 do not just have a single value but consist of a temporal vector which records all the values it has ever had along with the times at which it had those values. With this it is possible for cells to refer not only to current values but to past values and hence enables animation to take place by a cell referring to its own past. Another interesting consequence is that "time travel" is possible where the entire system is reverted to a previous state or where changes to the past can be made with the consequences propagated immediately to give immediate feedback, something which has been called *steering* [Burnett et al., 2001]. It is the liveness of the system which allows for steering whilst keeping the system consistent.

A language which provides further insight into the issue of user-defined types and for providing structure in an end-user friendly way is Self [Ungar and Smith, 1987]. The motivations for developing Self are given in [Smith et al., 1995] where the authors say that:

> *Programmers are human beings... they also need things like confidence, comfort and satisfaction – aspects of experience which are beyond the domain of pure logic.*

One of their key aims was to "integrate the intellectual and experiential sides of programming" [Smith et al., 1995] by providing a consistent and malleable world, not too dissimilar to the notion of a *plastic software environment*, to support exploratory programming [Ungar and Smith, 1987]. As experience and direct manipulation were fundamental ideas the authors decided to devise an environment based upon *prototypes*. Prototype-based object-oriented programming is an alternative approach to class-based object-orientation where new objects are constructed either ex-nihilo or by copying (*cloning*) from an existing concrete instance instead of from an abstract class blueprint [Burke, 2005]. The idea being that this prototype approach "corresponds more closely to the way people seem to acquire knowledge from concrete situations" [Lieberman, 1986], with the focus on the word concrete, something which also appears as

an important concept in the development of Forms/3.[5] It is interesting how the *type definition forms* in Forms/3 seem to be a hybrid concept between class and prototype since these forms are used like a class but can be copied like a prototype [Burnett et al., 2001].

Five key reasons are given in [Ungar and Smith, 1987] for using prototypes instead of classes, these are as follows:

1. Simpler relationships with only a single "inherits from" concept which allows for simpler inheritance hierarchies.

2. Creating by copying where copying is a simpler metaphor than instantiation.

3. Examples of pre-existing modules, allowing a user to examine a typical representative as opposed to attempting to make sense out of a description.

4. Support for one-of-a-kind objects which allows individual objects to be customised directly and independently.

5. Elimination of meta-regress which is a conceptually infinite problem where a class is an instance of a meta-class and so on.

Something considered an important characteristic of Self is its support for live editing, an important EUD principle. According to the authors Self provides "an unusually direct interface to such live changes" [Smith et al., 1995], in part due to the close match and integration between the language and the environments user interface called Morphic [Maloney, 2000]. The directness and liveness of Self depends not only on the purity of the object-oriented language but also on the direct manipulation capabilities of Morphic and its *meta-menu* to make live changes, which creates an "experience of programming that can be learned more easily [by end-users]" [Smith et al., 1995]. The *meta-menu* is available for all graphical objects, called *morphs*, which allows those

[5] "Immediate feedback is facilitated when concrete objects are present in the programming environment [and so] concreteness is a goal as well" [Burnett et al., 2001].

objects to be changed. The meta-menu also provides an *outliner* menu item to show and edit language properties of the object such as slots which include methods and properties.

Not only is Self one of the first and most significant prototype-based languages to focus on the user's experience, but it was also the first interactive pure object-oriented language to have good performance and good responsiveness [Hölzle and Ungar, 1994]. Some of its optimisation strategies have gone on to be used in more well known languages such as Java. What this has done is show that end-user friendly languages do work sufficiently well for real applications.

The final system to be looked at here is Subtext developed by Jonathan Edwards at MIT [Edwards, 2005] and its more recent incarnation Coherence [Edwards, 2009]. Subtext has much in common with both Self and Forms/3, both of which influenced its development. It has spreadsheet like formula characteristics as well as prototype-based cloning as a core concept. Edwards is attempting to "transcend paper-centric programming" and reduce Norman's gulfs of execution and evaluation [Norman, 1998] by making the representation of a program the same as its execution[6]. This involves moving away from just text but at the same time not taking the usual visual programming approaches which have a tendency not to scale well and, according to Edwards, are still *paper-centric*.

With Subtext the programmer is working directly with a tree structure that is both the code and the data, using a suitable graphical interface. Nodes form structures and at the leaves there can either be an empty structure called an "atomic value" or a reference which is much like a spreadsheet formula that returns a node (which may be an "atomic value"). Functions are "structures that react to change" [Edwards, 2005] and use the exact same principle as Core Forms/3 [Djang et al., 2000] where there is a structure with subnodes (cells in Core Forms/3) for the parameters and another subnode for the result with a formula in the result that describes the function. Spreadsheet-like

[6]This similarity of representation in Subtext relates well to the notion of *computation by navigation* introduced in chapter 4 of this thesis.

dependency maintenance is then used to give immediate feedback in response to any change and so Subtext supports the EUD liveness principle.

In some respects, therefore, Subtext is a prototype-based environment that has not embraced the object-oriented paradigm. It is not based on message passing nor does it have imperative methods. Whereas OO merges state and behaviour and Self in particular focuses almost entirely on behaviour, Subtext has achieved the opposite by focusing entirely on state with all behaviour actually being a form of state maintenance. Even function calls are done by copying. In summary then, Subtext is making use of the prototype-based approach to achieve a form of programming that involves direct manipulation rather than indirect textual manipulation. It is perhaps closer to Empirical Modelling than the object-oriented Self language is due to its focus on state.

Edwards has since gone on to develop a new language called Coherence which is based on his concept of coherent reaction [Edwards, 2009] that is much more dependency-like than Subtext. Unfortunately he has recently discovered that his approach is non-deterministic so has abandoned that project (attempting to make it deterministic by moving away from dynamic prototypes to static classes instead) [Edwards, 2010]. Despite this it will be discussed here as an example of a prototype-based and dependency-based language combined.

As with Subtext the Coherence language is based upon a dynamically typed mutable tree and nodes within the tree are created by copying from existing nodes. It uses inheritance mechanisms similar to those found in Self where if a field does not exist then it searches up the inheritance hierarchy to find it. Fields can be defined with a derivation expression which is lazily computed when needed to calculate the value of that field. It is described as being like a formula in a spreadsheet cell and is guaranteed not to produce side-effects. What is unlike a spreadsheet, however, is that derivations are bidirectional so changes to the derived field propagate back to the variables it was derived from. This has been called the reaction by Edwards. It is this that ultimately creates the problem of non-determinism because it is a constraint satisfaction problem,

or, as Edwards states, it is actually a problem of constraint discovery [Edwards, 2009]. Coherent reaction then was an attempt at performing constraint discovery without either reducing the expressive power of the language or involving the programmer. Edwards' higher goal was to deal with the problem of imperative programming where it is up to the programmer to orchestrate the exact order in which all events takes place. It is significant that he chose both the prototype-based and spreadsheet style approaches to achieve this and his justification is given with an example demonstrating the difficulties of execution order in imperative languages. It comes back to ultimately attempting to build a model of coherent state that can deal with change rather than focusing on behaviour.

2.1.4 Guidelines

From their work on AgentSheets, an EUD environment, Repenning and Ioannidou identified thirteen design guidelines for end-user development environments [Repenning and Ioannidou, 2006]. All of these guidelines are, to varying degrees, applicable to a plastic software environment and so will be utilised in the design and development of the new tool. Table 2.2 shows these guidelines.

Of particular relevance is the mentioning of domain-oriented languages and meta-domain orientation. These two concepts appear prominently in Empirical Modelling in the form of definitive notations and an underlying definitive language to bring them all together. Similarly, support for incremental development, decomposable test units and multiple views would be vital for plastic applications and also relates well to Empirical Modelling principles introduced later in this chapter. The fact that these guidelines and the EUD principles fit well with EM is no surprise as both are looking at the same problem but from different angles. Whilst EUD is coming from traditional programming and moving down towards end-users, Empirical Modelling, as will be explained, is coming from the concrete realm of experience and attempting to move up towards programming.

1	Make syntactic errors hard
2	Make syntactic errors impossible
3	Use objects as language elements
4	Make domain-oriented languages for specific EUD
5	Introduce Meta-Domain orientation to deal with general EUD
6	Support incremental development
7	Facilitate decomposable test units
8	Provide multiple views with incremental disclosure
9	Integrate development tool with web services
10	Encourage syntonicity
11	Allow Immersion
12	Scaffold typical designs
13	Build community tools

Table 2.2: EUD Guidelines [Repenning and Ioannidou, 2006]

2.2 Empirical Modelling

The Empirical Modelling (EM) research group is a small group of researchers and students based at the University of Warwick in the UK [EM Website]. The aim of the group is to explore and develop a conceptual framework and associated tools for informal modelling activities that are grounded in experience. Over the years a considerable number of these models have been developed with their tools by various undergraduate and postgraduate students at Warwick [EM Projects]. Their tools have often been used for educational purposes [Harfield, 2008], learning about problems [Care, 2006] and computing [Beynon et al., 2000a]. This section will cover the key concepts of EM, the existing tools and some example models and application areas. The conceptual framework and some of the concepts used in their tools provide an excellent foundation for plastic applications.

2.2.1 What is Empirical Modelling?

The word *empirical* should be well understood as meaning "derived from experience and experiment" and gives a clue to what Empirical Modelling is about. EM takes this notion of experience and experiment right to the core of its conceptual framework to describe a new modelling approach based "firmly on direct, *living* experience rather than formal representations" [Beynon, 2011, p.1]. It is argued that experience is primary and comes before any theory and formal representations can be developed, relating strongly to William James' notion of *radical empiricism* [James, 1912/1996; Beynon, 2007] where to know something is to experience an association between one aspect of our experience and another.

One of the motivations behind the inception of EM comes from trying to develop "theoretical frameworks that do justice to [the] practice" of programming by taking account of first and second factor concerns [Smith, 1987]. Smith's first factor corresponds to the execution of a program and traditional theory, whilst the second factor is about the (human) interpretation of programs. It is the gulf between the formal and informal

worlds. Beynon and Russ discuss these two factors in [Beynon and Russ, 1992] where he says that "an adequate theory of computation must allow a high degree of interaction between these two factors". It is noted by Beynon and Smith that computer science only talks about the first factor and that "curiously" the semantics of a program do not consider the second factor, that of interpretation. The consequences of this lack of concern for the second factor are eloquently put by Black:

> *"the drastic simplifications demanded of success of the mathematical analysis entail serious risk of confusing accuracy of the mathematics with strength of empirical verification in the original field"* [Black, 1962] as cited in [Beynon, 2011]

Focusing only on the abstract mathematical world and ignoring the concrete reality we inhabit will likely mean that any model, program or theory developed in such a way will be suspect when it comes to being interpreted. In practice the concrete is not ignored:

> *"In devising a mathematical model of a tree, the mathematician adopts a constrained way of observing features relevant to a functional objective but may first need to identify suitable features and patterns of behaviour derived from exploratory experiment"* [Beynon, 2011, p.5]

If the word "mathematical" is replaced with "computer" and "mathematician" with "programmer" then the above statement by Beynon accurately describes the need for a software development process with requirements (a functional objective) that need to first be formulated. Exploratory experimentation is a term used by Steinle for experimentation that takes place before any well formulated theory exists, in contrast to experimentation that is "theory-driven" [Steinle, 1997].

> *"Exploratory experimentation, in contrast [to theory-driven experimentation], is driven by the elementary desire to obtain empirical regularities and*

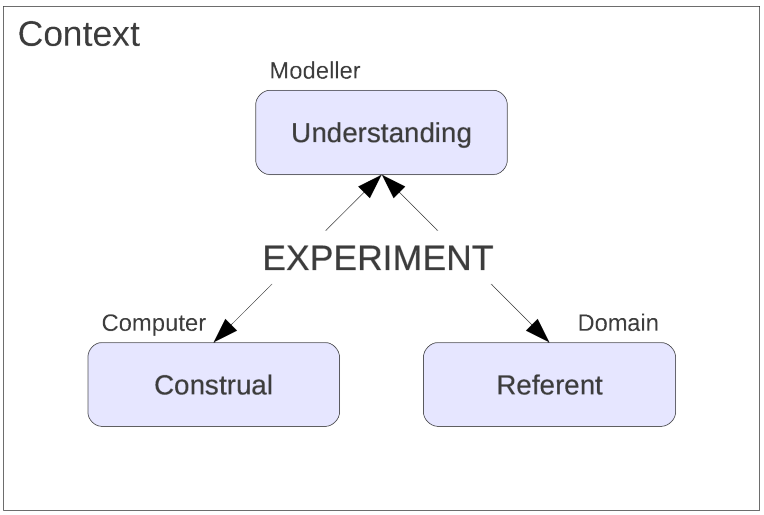

Figure 2.1: Original EM Conceptual Diagram

to find out proper concepts and classifications by means of which those regularities can be formulated." [Steinle, 1997]

Exploratory experiments can be found throughout the scientific community, with a well documented and interesting example being the experiments done by Faraday in trying to develop a theory of electromagnetism. Gooding provides an account of Faraday's work which attempts to discover how Faraday was able to gain important understanding through the use of physical, metaphorical models of otherwise invisible forces [Gooding, 1990]. Faraday uses exploratory experimentation to eventually develop a theory. It is clear from this, as well as from Black's comment, that everything is, and should be first and foremost, grounded in experience. This is certainly the tenet of Empirical Modelling which has borrowed ideas such as that of *construal*[7] from Gooding's work on Faraday.

So Empirical Modelling is attempting to develop a framework for pre-theory exploratory modelling which enables the modeller to identify "empirical regularities" and gain sufficient understanding to construct a formal account. Experiment is to be used to

[7] A construal is a provisional, personal sense-making entity [Beynon, 2011, p.1] and originated in the work of David Gooding [Gooding, 1990] regarding the modelling activity of Faraday.

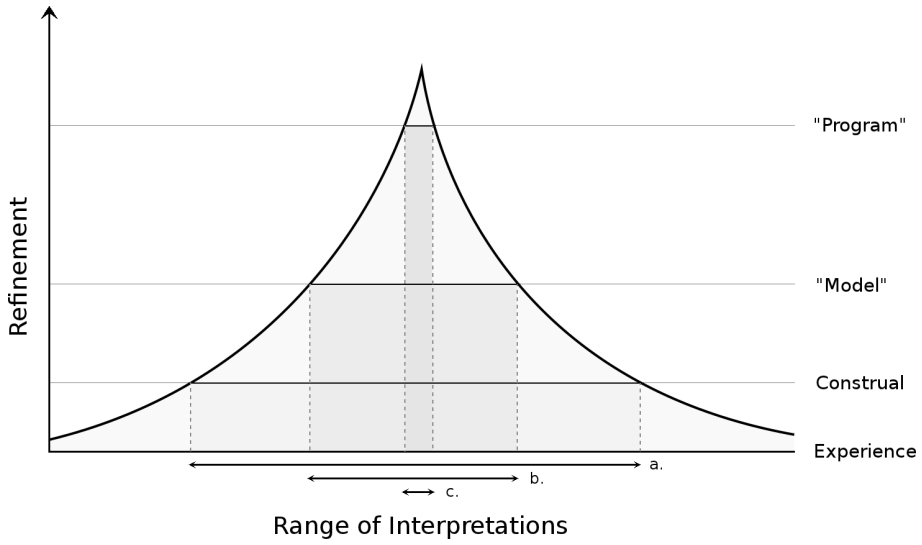

Figure 2.2: Refinement from experience to "program"

bridge the gap between immediately experienced and circumscribed behaviour [Beynon, 1994]. Figure 2.1 shows how this has been conceptualised as a process involving four components which co-evolve in a live fashion to construct and refine models whilst allowing the modeller to gain understanding [Beynon, 2011].

The diagram in figure 2.2 is intended to illustrate how, in EM, an artefact is refined from being initially an open-ended experience to being a "program" with restricted interpretations and *ritualised interactions*[8]. The curve shows, in a much simplified way, the boundaries of what the modeller considers as meaningful interactions and interpretations at a particular stage of refinement. As the artefact becomes increasingly refined (moving up the y-axis) through a process of exploratory experimentation the range of possible interactions and interpretations becomes increasingly restricted, shown by a,b and c. Of course, the process is considerably richer than indicated in figure 2.2.

> *"the model migrates... from provisional to assured, subjective to objective, specific to generic, personal to public. To support this migration, the model has to be fashioned, via interactions both initiated and automated, so that*

[8]See Harfield's PhD thesis for more on ritualised interactions [Harfield, 2008, p.33].

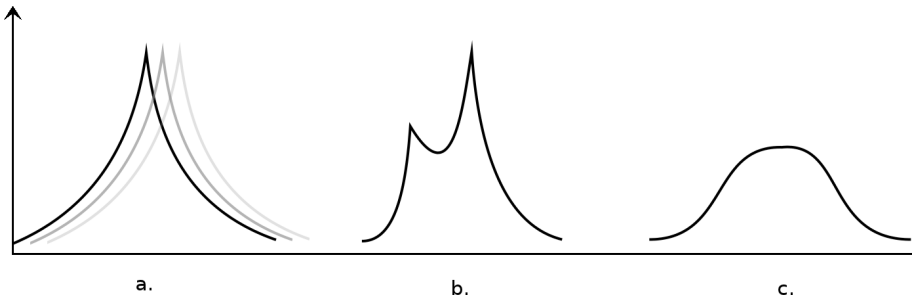

Figure 2.3: Problem Shapes

> *the views of many agents - human and non-human - are taken into consideration. This process of construction is in general open-ended and never absolutely resolved."* [Beynon, 2011, p.17]

An analogy of devising a walk has been used in [Pope and Beynon, 2010b] to explain the evolution of an EM artefact. Initially any walk plan is entirely unformed and the possibilities are almost unrestricted, which corresponds to being at the experience level in figure 2.2. Gradually the walk planner will decide on a region and then maybe a specific mountain to walk on. This is a refinement which moves up the diagram to perhaps be considered a *construal* or "model". The process continues until the planner has a precise set of instructions for paths to follow and this would be at the level of refinement expected of the "program" level. What this analogy shows is how initially any plan is provisional, subjective and personal but will be refined by an exploratory process until it is assured, objective and public. The walk plan is initially at the mercy of the planner's imagination (within certain limits) but eventually becomes something which can be described to others and followed in an objective fashion.

Figure 2.3 shows how in practice the curve showing the boundary of meaningful interactions and interpretations is more complex than that of figure 2.2. a) shows how this curve, which corresponds to the environment, context and modeller's goal, can change with time. This would mean that a well refined artefact may, without itself changing, become incorrect with respect to some criteria. b) is a situation where

it is possible to refine an artefact to a point and get "stuck", requiring a degree of backtracking to a previous and less refined state in order to explore and experiment a different way. Finally, c) shows a scenario where refining to an assured, objective and public artefact may not be desirable. What is important then is for an ability to move smoothly from construal to model to program and back again. The categories of construal, model and program are blurred concepts attempting to name specific stages of a continuous process.

An important note to make is that the terms "program" and "model" as used in figure 2.2 are not identical to the traditional concept of program and model. A traditional program will typically have fixed functionality to enable optimisation, whereas the EM "program" still has a flexible functionality (possibly at the expense of not being optimised). Unfortunately perhaps, computer science and the software industry have widely assumed that problems must be abstracted before they can be solved, with increasing layers of abstraction being the solution to complexity. Computer Science has stuck with the first factor, ignoring the second and so results in inflexible programs.

> *"Whereas conventional modelling uses abstraction to simplify complex phenomena, EM generates an interactive environment in which to explore the full range of rich meanings and possible interpretations that surround a specific ... phenomenon"* [Beynon, 2011, p.3]

One of the major challenges, besides developing an appropriate conceptual framework, is finding ways of supporting the migration described above. A more in depth discussion of the philosophical and conceptual aspects of EM can be found in [King, 2004; Beynon, 2007; Beynon and Russ, 1992; Beynon, 2011]. These conceptual issues are vital for providing a solid foundation so are discussed in chapter 6. However, the focus of this thesis is to explore ways of achieving the migration from construal to program, something not currently practical with existing EM tools, as will become clear, but which is believed to be possible with the EM conceptual framework in general.

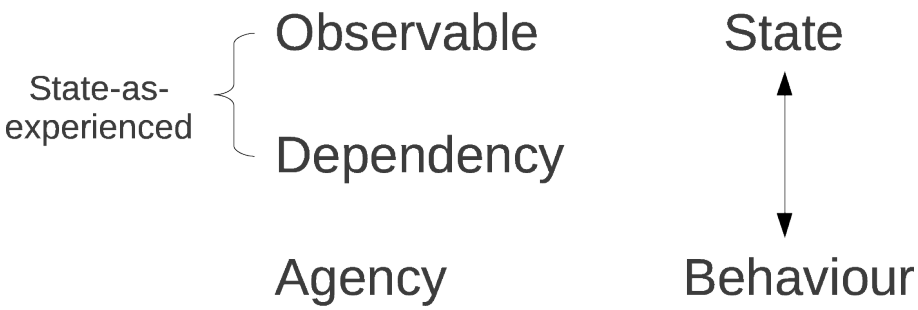

Figure 2.4: EM ODA Framework

2.2.2 The Principles

One of the key objectives of EM is to provide a solid conceptual framework and philosophical account of this kind of exploratory, experiential modelling activity on a computer. An experiment involves the observation of and interaction with state and so Empirical Modelling principles are "fundamentally concerned with modelling state" [Harfield, 2008, p.31]. Since the state being modelled is "empirically apprehended" and is somewhat different to traditional program state, it has been called *state-as-experienced* [King, 2004, p.30]. In addition to modelling state the notion of agency, the observation and interaction, needs to also be considered and so becomes the second central concept.

The construction of artefacts that relate to *state-as-experienced* in the referent is the process being described by figure 2.1. In the diagram there is only a single modeller involved, but there is little reason not to consider multiple human agents being involved and work to that effect can be found in [Sun, 1999; Chan, 2009; Beynon and Chan, 2006].

The concept of *state-as-experienced* can be split into two: observables and dependencies. The motivation for this split lies in the fact that we not only observe quantities or qualities of experience but also the associated relationships between them. Therefore there are three core concepts in Empirical Modelling which are well expressed in [King, 2004]:

- Observable: "something which has a value or status to which an identity can be

ascribed". There are no restrictions on what this might be and certainly does not need to be numerical. It is not a mathematical variable due to its concrete nature, but may have some connection to variables in traditional programming languages.

- Dependency: "a relationship between observables such that interaction with one observable leads indivisibly in our experience to a change in the other". These relationships are what allow changes to propagate and crucially are what enable experimentation to occur. They are the "experientially-mediated" associations which, from radical empiricism [James, 1912/1996], provide meaning.

- Agency: "an agent is projected on to the referent as something that can change the state of the model in some way by manipulating observables and the relationships between them". Each agent may have a different view and interpretation of the model of state. Agents may be human or non-human, there may be one or many and they may also act concurrently [Beynon, 1997a].

The first two concepts enable a rich model of state to be developed which remains coherent in the presence of change and also, due to the way it is developed, remains meaningful to the modeller. A vital component not often highlighted in the practical aspects of EM is the significance of an "observational context" [Beynon, 1994]. What observables and relationships are important to a given agent at a given point in time will depend on a potentially shifting context of observation. An example given in [Beynon, 1994] is of a Newtonian model for projectile motion. In this model certain observations such as air-resistance are ignored initially but at some future date the context may change and such observations may then become important. A classic example given by Beynon in lectures on Empirical Modelling is a line drawing model of a filing cabinet (figure 2.5) which may also be interpreted as an LCD digit [Beynon, 1990]. Which interpretation is to be taken will influence what observables and relationships exist and what agent actions may be meaningful (it makes sense to open a filing cabinet but not to open an LCD digit). At any time the context could shift from filing cabinet to LCD.

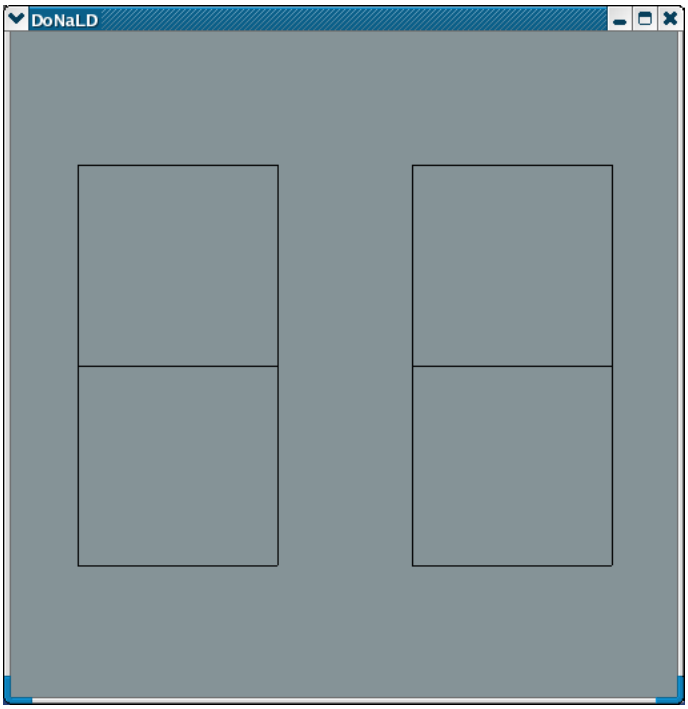

Figure 2.5: Filing Cabinet and LCD Digit

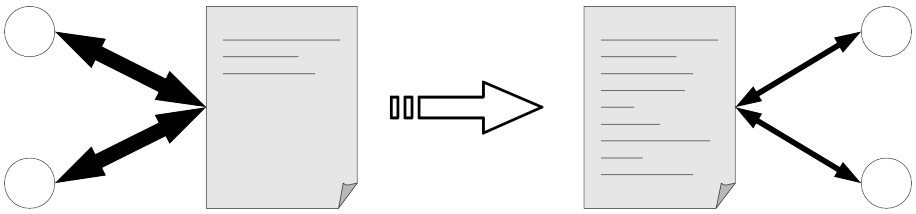

Figure 2.6: Agents interacting with a definitive script. As the script matures agent interactions are restricted.

State-as-experienced (observables and dependencies) is embodied in a *definitive script*, a concept that has been with Empirical Modelling since Yung developed EDEN [Yung, 1990] and which has been explored in depth in [Rungrattanaubol, 2002]. All interaction with the model is conceived as making redefinitions in the script. A definitive script may consist of a collection of scripts, each of which may use a different *definitive notation*[9] that is task-specific. Figure 2.6 shows how agents interact with scripts and that over time these interactions become *ritualised* and restricted.

2.2.3 Current Tools

Whilst EM is primarily a conceptual framework there has been considerable tool development to support such a modelling activity on a computer. The only Empirical Modelling tool currently in use is Tkeden, originally called EDEN[10] by Edward Yung who first developed it [Yung, 1990]. EDEN was intended to be a "general-purpose language supporting definitions" [Yung, 1990, p.6] which borrowed much from spreadsheets and C. It is useful to note that EDEN was developed around the same time as both Self and Forms/3 discussed previously. Since its inception, EDEN has been extended by many developers and used by hundreds of students and academics [EM Projects]. Ashley Ward gives a detailed up-to-date picture of the EDEN tool in chapter 4 of [Ward, 2004].

Additional tools have been explored, including the Abstract Definitive Machine (ADM) by Mike Slade [Slade, 1990], Definitive Assembly Maintainer (DAM) by Richard Cartwright [Cartwright, 1999] (along with a Java version called JaM and subsequently JaM2) and finally the Definitive Object State Transition Engine (DOSTE) [Pope, 2007] which was developed by myself as Empirical Modelling coursework. The original DOSTE is not the same as the work given in this thesis but was the origin of some of the concepts. Again, these tools (except DOSTE) have been reviewed in detail by Ward in [Ward, 2004].

[9] A *definitive notation* called ARCA was the origin of Empirical Modelling [Beynon, 1985] and so this concept has been involved since the beginning. It is intended to be a specific algebra for a particular set of problems, in this case Cayley Diagrams.

[10] Engine for DEfinitive Notations.

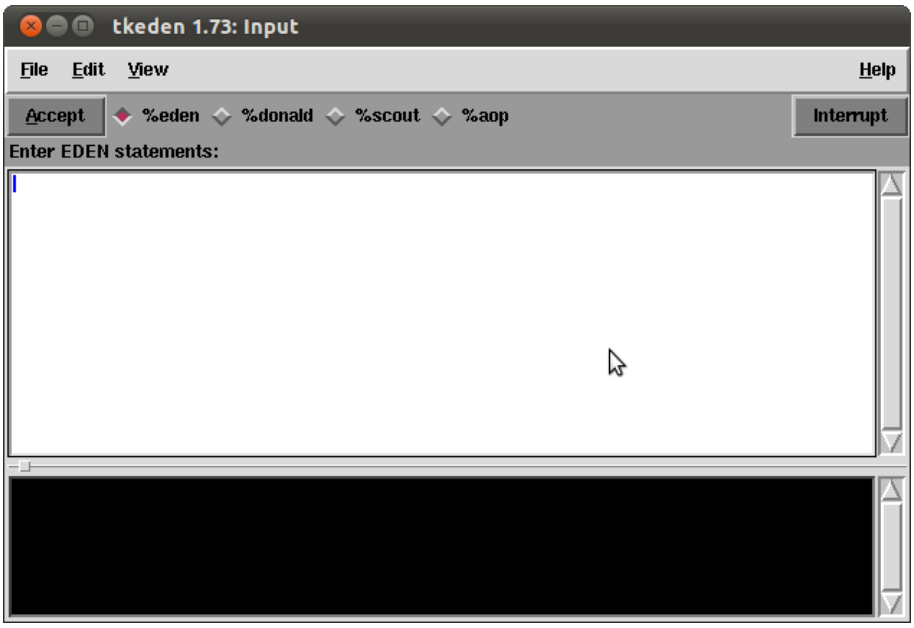

Figure 2.7: Tkeden Input Window

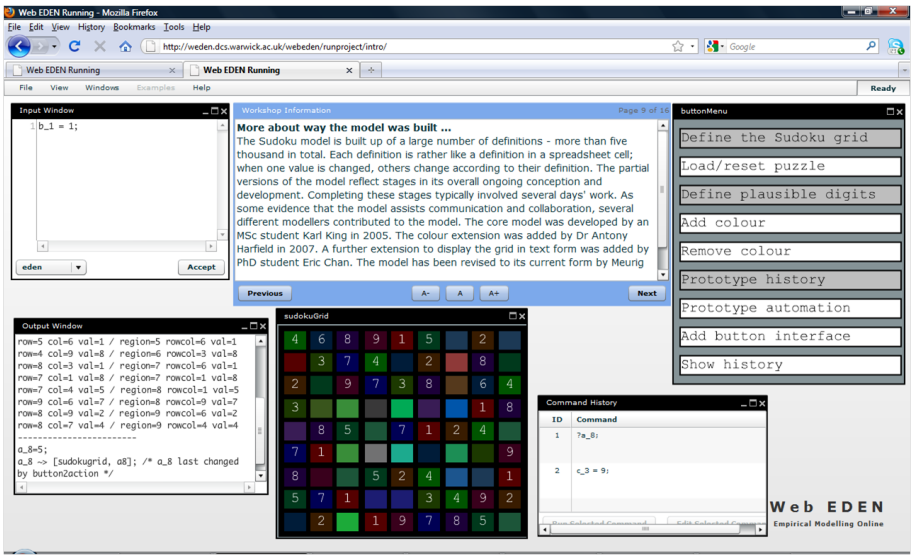

Figure 2.8: Web interface to Tkeden developed by Richard Myers

For the purposes of brevity only the EDEN tool will be discussed here, although the ADM with its object-like concepts has been influential. Eden, the definitive language of the EDEN tool, uses a less strict C-like syntax with the addition of appropriate syntax for describing dependency definitions and dealing with agent actions. A version of EDEN called *tkeden* provides a simple interface as shown in figure 2.7. More recently a web interface to *tkeden* has been developed by Myers [Harfield et al., 2009] and is shown in figure 2.8. The key difference between the various EM tools is how they interpret and implement the three core concepts. Each tool takes a different approach, especially true for agency. In EDEN the ODA framework has been implemented as follows:

- Observables: The interpretation given to observables in EDEN is that of a flat space of variables with a specific and limited set of available types along with a C-syntax identifier. The typing is not rigid as no observable gets declared to be a single type but can change type at any time. The available types are: integers, reals, strings and lists. There has been some work to extend the original EDEN so that observables can be grouped into something called a *virtual agent*. This virtual agent feature has not proved successful or reliable and so remains unused in the most recent models.

- Dependencies: Expressed as functional like formulas using a C-syntax and associated with a specific observable in a way that is similar to a cell being given a formula in a spreadsheet. All references to observables in the definition are considered to be a dependency and so when any of those set of observables changes this particular formula/definition is marked as out-of-date and re-evaluated when next accessed. All of this dependency maintenance occurs indivisibly to the modeller and any agents.

- Agency: There are two distinct forms of agency to consider in EDEN. The first is the modeller (a.k.a. the human user). The modeller can interact with a model in a variety of ways including a textual input window (cf. figure 2.7) or with mouse

clicks. In the textual input window they may enter script in one of several different notations depending on the domain of interaction or observation. Such notations include the basic Eden notation and notations for line drawing (DoNaLD), window management (SCOUT), 3D graphics (Sasami) and relational databases (EDDI) as well as a newer notation for constructing custom notations (AOP) [Ward, 2004]. In addition to the modeller there is some scope for automated agency. Automation is achieved through the use of triggered procedures which can request to be called whenever a specific set of observables has changed (either directly or via some dependency maintenance). Such triggered procedures can then use conditional statements, loops and other similar constructs to observe and make changes to the state of the system by changing values of observables or giving them new definitions.

Something that needs further elaboration is the use of multiple notations. The purpose of this is to provide different algebras, in the form of data structures and operations, that are specific to a particular domain such as line drawing. All the notations convert down to the underlying Eden language to build up the required sets of observables, dependencies and agents which correspond to the more abstract concepts in these "higher" notations.

2.2.4 An Example Model

To illustrate the Empirical Modelling process and show the concepts in action, the digital watch model will be briefly discussed. This model was originally developed in *tkeden* by Beynon in 1992 based upon a state chart by David Harel in [Harel, 1988] of a digital wrist watch. Over a period of eight years four different people, including Beynon, were involved with this model, extending, refining and revising it (cf. figure 2.9) [Fischer and Beynon, 2001; Roe et al., 2001; Beynon and Cartwright, 1995; Roe, 2003]. Initially the model involved only the state chart and a digital display, developed over a period

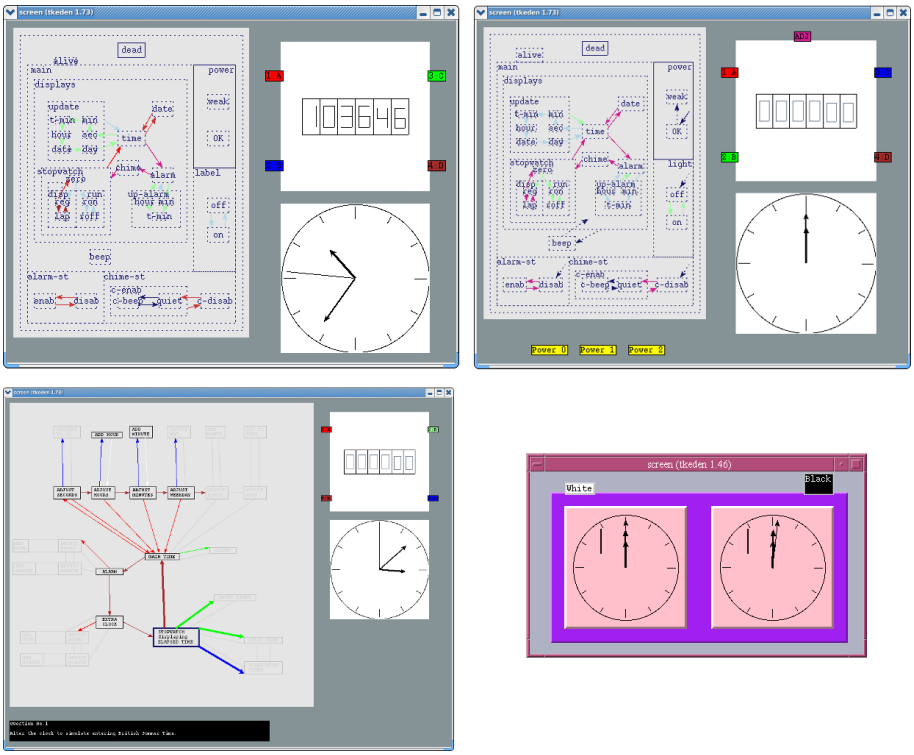

Figure 2.9: Evolution of the Digital Watch. Top-left: Cartwright 1995, Top-right: Fischer 1999, Bottom-left: Roe 2001 and Bottom-right: Cartwright's Chess Clock 1995

of about 3 weeks using a continually running program on an office workstation.[11] The model was developed incrementally by adding in small groups of definitions and agents to add some new visual component or agent behaviour. Interestingly Beynon states that "it was sometimes useful to develop portions of the script independently, or to trace problems by extracting pieces of the script and exercising them in isolation" [Beynon and Cartwright, 1995], which relates well to some of the EUD guidelines and indicates the flexibility of the tools. Beynon's resulting model consisted of 2000 lines of script with around 1000 definitions and was then used to help teach EM to MSc students at the time.

Following this, students independently and without much help from Beynon, were able to experiment with the model to learn how it worked and subsequently extend it in various ways. Some of these alterations were examples of a radical shift in *observational context* that involved adapting the model to an entirely new purpose, such as Cartwright's Chess Clock shown in figure 2.9 [Cartwright, 1995]. Other changes involved the addition of new buttons and functionality to, for example, allow the date and time to be changed.

Changing the model is a relatively simple activity. The modeller can enter new definitions, written in one notation or another, into the *tkeden* input window and click "accept" to see the consequences of that change immediately. Such a change might be to make the hour hand of the analogue display depend upon the minute hand so that if the minute hand is changed the hour hand also changes by dependency [Fischer and Beynon, 2001]:

$$\text{angle\_hour\_hand} = (\text{angle\_min\_hand}/2\pi) * (\pi/6)$$

Other changes may be made to the visual representation, such as the colour of a button. Not only do these changes incrementally develop the model, they also allow for experimentation and testing. Normally, in the more developed versions of the model,

[11] Except for the occasional crash, "but several days typically elapsed between such events" [Beynon and Cartwright, 1995].

agents are responsible to operating the various watch mechanisms. However, this can be interfered with by the user to, for example, simulate a malfunction of the alarm or battery. Such interference is outside of the preconceived pattern of interaction [Fischer and Beynon, 2001]. This kind of experimentation serves several purposes: to gain understanding of the model, to learn something about the referent or to check how well the model matches with the referent (does a particular interaction cause a similar effect in the model as it would in a real watch). Unlike some EM models the digital watch model is based upon a well understood device and so is not a truly creative and exploratory endeavour unless it was to look at new kinds of watch design or other changes of context.

2.2.5 EM and Software Development

There has been considerable work on using Empirical Modelling as a new way of developing software, with EDEN being partly developed under the title of "a Paradigm for Exploratory Programming" [Yung, 1993] and Ness who looked at "Creative Software Development" [Ness, 1997], along with other work by Beynon et al. [Pope and Beynon, 2010b; Beynon et al., 2008; Beynon, 2011]. A common response of those from industry (and from students) who are introduced to EM is that it provides a means of requirements elicitation and perhaps prototyping but nothing more. By taking this view they are missing the point and the potential of EM.

Traditionally developed software relies on "pre-existing ... theory or established empirical knowledge" [Beynon, 2011, p.1] as "formal representation draws on previous experience" [Beynon, 2011, p.18] which may then be appropriately abstracted. It is only possible to generate abstractions once sufficient understanding has been obtained. In contrast, EM looks to support not only the traditional software (by in principle supporting abstraction) but also software where there is no pre-existing theory or pre-computer precedent by allowing for experimental exploration that is yet to be abstracted. In effect it is allowing computer science to be an experimental science [Milner, 1986].

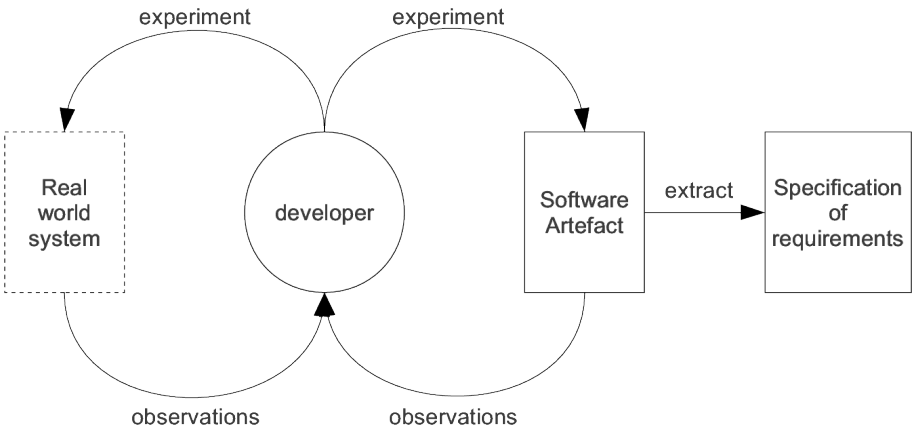

Figure 2.10: EM Software Development Process, based upon diagram in [Beynon and Russ, 1995]

So EM does begin before any traditional requirements stage and does involve a form of prototyping, however, there is little reason that, with appropriate tools and framework, these models cannot evolve into applications without a big leap from requirements to design to implementation. The diagram in figure 2.10 shows the EM development process as conceived in 1995. What is significant is that a "specification of requirements" is extracted from a model to then generate a fixed functionality program, but nothing beyond that has been elaborated. The process at the time tried to integrate the requirement, specification and design phases of a development, but not the final implementation which was to be done by some unknown transformation from model to program [Beynon and Russ, 1995].

There has been little progress on this since 1995 with most work becoming centred on using *tkeden* or mapping the conceptual framework onto specific domains such as product design [Ness, 1997] or educational tools [Harfield, 2008]. There have been attempts made (but not well documented) on an automatic translation process from model to program. Some work was done by Sun on distributed modelling and software development which involved adapting *tkeden* to produce a distributed version called *dtkeden* [Sun, 1999]. Recent work on EM and constructionism also gives some in-

sight [Beynon and Harfield, 2010]. The most successful models in software development terms are the temposcope [Beynon et al., 2000b] which was briefly used as a finished product, and colour sudoku [Beynon and Harfield, 2010] that has had widespread appeal. Full transition from model to program has been a goal for some time but as yet not achieved. Scaling up is perhaps the limiting factor.

2.3 Miscellaneous Technologies

Despite the existence of structured data models, such as relational databases, there is a need to represent data that cannot be constrained by such strict pre-designed structures. The World-Wide-Web is perhaps one of the key motivations for developing the notion of *semi-structured* data (also called *self-describing* data) as it is difficult to see how this could be constrained by a schema [Buneman, 1997]. Another issue apparent from the beginning is the inability of users to browse a traditional database without writing a query which requires knowledge of the schema [Buneman, 1997], making exploration of the data difficult. The approach taken by researchers and industry for dealing with semi-structured data is to represent the data as an edge labeled graph. Today the most well known and exceptionally popular approach is that of XML. The notion of semi-structure will become relevant when considering ways of improving EM tools (cf. §3.4.1).

Most often based upon XML, dependency injection is a way of configuring components and attributes of a program by using an external description which links the components together [Chiba and Ishikawa, 2005]. For example, XML can be used with Java to connect various objects together at load-time rather than embedding those connections in the source code. This enables the reconfiguration of components without recompilation of the program and is an attempt by developers to reduce the dependencies hard-coded into the application, enabling component reuse. These dependencies are then available for modification by the developer, although not usually in a live fashion.

The use of dependency as found in EDEN can also be seen in commercial products other than spreadsheets. Microsoft's Windows Presentation Foundation (WPF)

includes the concept of *dependency properties* [Cox, 2008; Harfield, 2009]. Dependency properties enable simple connections between properties to be made but it is difficult to describe more complex relationships. Additionally, WPF dependency properties are not interactive but get compiled into the application, there is no possibility of changing the dependencies live. Flex is another technology that is also employing the use of dependency in describing interfaces.

Chapter 3

Enabling Plastic Applications

Within this chapter the exact nature of the problem being addressed by this work is explored. In chapter 1 the concept of *plastic applications* was introduced and identified key problems that need to be overcome on a grand scale. Chapter 2 then introduced Empirical Modelling along with EUD and other technologies that are of considerable interest with respect to achieving the *plastic applications* aim. This work will focus on adapting the existing Empirical Modelling concepts and tools in order to fulfil that goal and to do this it is necessary to give a critique of EM. Once specific problems have been identified the focus will turn towards industry and EUD approaches to see how they may resolve the problems found in EM. The result of this will be a specific set of questions and ideas that are to be the focus of the remaining chapters in this thesis. It is in this chapter that the problems with EM are identified with solutions proposed and the specific research questions being expressed.

3.1 Empirical Modelling and Plastic Applications

Plastic applications have been introduced by this work to classify applications which can be freely moulded by an end-user whilst also being capable of solidifying into what might appear to be a more traditionally developed application. Perhaps another phrase

is *flexible functionality* programs as opposed to *fixed functionality* programs[1]. Empirical Modelling has developed a similar concept in its notion of "program" which is something that has evolved from a construal and has become "constrained" through *ritualised* and restricted interactions. An EM "program" is, as already discussed in §2.2.1, not the same as a traditional program in that it supports *flexible functionality*. In this work the EM notion of "program", when it has a particular resulting functional objective, will be taken to be the same as the concept of a plastic application. The EM idea of "program" has only become clear recently as a result of this work on plastic applications. Previously to get to a program from an EM model it was assumed that a translation process was required (cf. figure 2.10). As a consequence there has been little to no prior research into how to transition from an EM model to a plastic application without a translation step. Whilst the EM conceptual framework has easily (and appropriately) been adapted to include this idea, there is little support in the EM tools for it and there are likely to be conceptual consequences yet to be identified. Despite this the Empirical Modelling conceptual framework is to be explored as a means of supporting plastic application development. In the remainder of the thesis the meaning of *program* will, unless otherwise stated, be the EM concept of "program" instead of the traditional interpretation.

In Empirical Modelling the migration from *construal* to program is considered as a change of context with a "different family of pre-engineered interactions and interpretations" [Beynon, 2011] but is always represented as "a net of observables and dependencies". This net will be known as the *Observable Dependency Network* (OD-net). The OD-net is developed over time from experience and so its meaning remains directly grounded in experience, consequently the program remains meaningful so long as the OD-net remains. So to enable plastic applications it is necessary to constrain interactions and interpretations of the OD-net by human and non-human agents but keep the OD-net in place at all times, allowing any restrictions to be relaxed. A tradi-

[1]Fixed and flexible do not refer to any run-time modularity mechanisms or similar techniques, but are much more radical in nature and directly involves end-users.

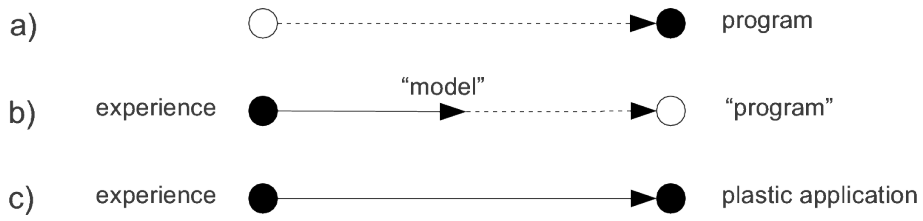

Figure 3.1: Transition from personal to public. a) shows traditional programs that have no such on-line transition. b) gives Empirical Modellings attempt and c) shows the ideal plastic applications result.

tional program does not have an OD-net, or any similar concept, and as a consequence remains isolated from experience which is perhaps why there are such difficulties in dealing with changes to requirements and also why formal approaches need to be taken to provide semantics. Looking back at figure 2.2 a traditional program is removed from the refinement curve by an abstraction process that is performed as a separate step. This step is usually performed to translate from requirements and design into an actual, and optimised, implementation on a machine. Such a step is contrary to the first EUD principle given in §2.1.1, that of being a *gentle-slope* system. It is also unnecessary since Self [Hölzle and Ungar, 1994] has shown how interactive applications can be optimised as needed and EM has given extensive evidence that rich models can be developed without abstraction being centrally important[2].

The Empirical Modelling process purports to support the full transition from experience to program, a reasonably recent development in itself. The reality is more like that shown in figure 3.1 where EM only partially supports the transition in practice. Much of the work on enabling EM tools and concepts to be used as a plastic software environment will relate to enabling the transition from "model" to plastic application.

[2]For example, colour sudoku [Beynon and Harfield, 2010; Harfield, 2007b], the temposcope [Beynon et al., 2000b] and the ant navigation model [Keer, 2010].

3.2 Dimensions of Refinement

To get a better handle on what is needed to support a plastic application it is necessary to revisit the dimensions of refinement first introduced in §2.2.1. The refinement process, as illustrated in figure 2.2, has at least four dimensions which together are a measure of the refinement of a software artefact. These dimensions are:

1. Personal → Public
2. Subjective → Objective
3. Provisional → Assured
4. Specific → Generic

It is the right-hand-side, the public, objective, assured and generic, which are the plastic application end of the refinement process and which need the most attention. Each of these dimensions has certain characteristic properties associated with them which give a form of specification for what any plastic software environment must enable.

Public A public entity needs to be communicated and shared among many people. Any experientially-mediated associations and observables in the artefact are to be understood by others in that they can also identify the same associations in experience. To share an experience there needs to be a common understanding, a common interpretation, perhaps by following certain conventions or by "embedding" specific interpretations into the artefact.

Objective The artefact must become independent from the individual modeller's emotions and thoughts by being fully realised as an actual artefact. It needs to be complete in the sense that no part of the model can remain only in the mind of the modeller, and also specific rather than fuzzy about what it is.

Assured A fixed, precise and perhaps formal interpretation is required. There may need to be certain guarantees, either by formal proof or by extensive testing, to show

that the application fulfils some goal. Again restrictions on interaction become important, as do security concerns if multiple users are involved. Resilience and robustness are also important.

Generic Instead of being about a specific concrete scenario the application becomes abstract and adaptable to more generic situations. The application can then be used for a broader set of problems than the original one which was used to gain better understanding but is now understood sufficiently well to be applied elsewhere.

What is perhaps unclear is the true role of abstraction and formal representations. Beynon claims that EM is complementary to formal approaches and that "it is possible to situate a formal representation within a context moulded from experientially-mediated associations" [Beynon, 2011]. Formality comes when stable patterns of interaction and interpretation can be appreciated *universally*. In EM, abstractions and formal representations have been given an auxiliary role which has resulted in such issues being under-explored in the tools, and in the framework. How in practice abstractions, restrictions and interpretations are to be given to an OD-net universally *at the level of a program* is unclear. The LSD notation [Beynon, 1986b] has been developed as a way of accounting for stable patterns of interaction and it was the intention of the ADM tool to *animate* the LSD accounts [Slade, 1990]. However, LSD was never intended to provide a "formal operational semantics" and "an LSD account does not lead directly to an executable model" [Beynon, 1997b]. What it does provide is a step towards abstraction and formality that needs to be capitalised upon far more than it is in current tools.

It is necessary to revisit the importance of the observational context (cf. figure 3.2). As stated by Beynon in [Beynon, 2011], the classifications of "construal", "model" and "program" are blurred and do not involve any real change to the OD-net but are instead a shift in observational context. It was highlighted earlier (cf. §2.2.2) that observational contexts are not well developed in the tools and so this is perhaps another concern to focus on to enable plastic applications. Without an adequate concept of

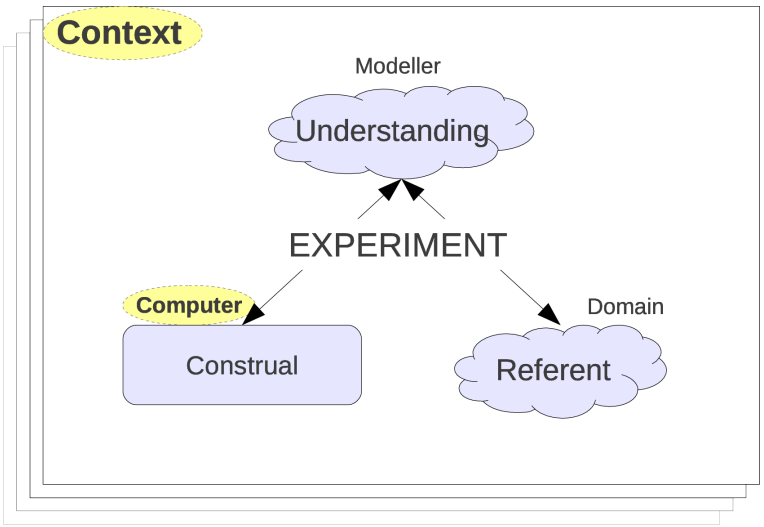

Figure 3.2: A need to improve support for *context* and for *construals* implemented on a computer

observational context it is difficult to see how the high degree of refinement in any of the four dimensions can be achieved. This relates to LSD in that an LSD account describes a particular observational context, one where particular patterns of interaction have been identified. An LSD account should not, however, be the only possible account and should not, as made clear above, have any effect upon the underlying OD-net so that other contexts may exist. Existing implementations of LSD do not obey this, partly because LSD itself encourages the OD-net to be developed with respect to its entities.

It is important to remember that there should be a smooth transition from one end to the other during the refinement process, and that critically such refinement can be reversed. The other end of the refinement process could also be improved. To be truly personal, subjective and provisional it is important that as many restrictions on interaction and interpretation are removed as is possible with a computer implementation, to allow for complete freedom. These improvements along with the ability to smoothly transition involve looking more closely at how a *construal* is supported on a computer (cf. figure 3.2).

3.3 Limitations of EM Tools and Concepts

Whilst EM is attempting to address the gulf between the informal world of experience and the formal world of programs, it has still been unable to fully bridge the gap in practice. The conceptual framework is believed to bridge this gap in principle, however, the tools that currently enable EM cannot realise this. Figure 3.1 shows the gaps between experience (informal) and programs (formal). EM has only been able to achieve a partial migration from construal to plastic application, due perhaps to its focus on the personal and experiential aspects.

To identify common problems a small survey of 20 WEB-EM[3] papers and models was conducted, and combined with my own knowledge/experience of many other projects and comments by previous PhD and MSc students who did not produce WEB-EM papers. A total of 20 relevant technical deficiencies were identified in the *tkeden* tool which relate to usability concerns and the concerns discussed in the previous section. These problems are discussed in this section.

3.3.1 Richness of Observables

In order to support construals and be faithful to the personal, subjective, provisional and specific nature of the software artefact in the early stages, observables need to remain as unrestricted as possible. What this means is that static types for observables are inappropriate and even the concept of type is too restrictive. EDEN is dynamically typed but with a limited range of types: integers, floats, strings and lists. These type restrictions have originated from the C language upon which EDEN is based. The EDEN list type is perhaps the most flexible in that it can be of any size, can contain heterogeneous dynamic types and does not name any of its components (all are accessed by integer index or list operations). These type restrictions do mean that more complex and non-standard kinds of observable (e.g. a shadow or bed) are difficult to represent

[3]Warwick Electronic Bulletin for Empirical Modelling, coursework for an MSc/MEng module at the University of Warwick.

in EDEN without resorting to abusing the list structure.

At the same time as saying that type restrictions are unwanted to better support construals, it needs to be recognised that structures and types do exist and that they are perhaps necessary to support the migration to a plastic application where abstraction and restriction are required. The EM solution is to use definitive notations and make it the job of the agents to observe specific types in a duck-typing[4] fashion, but to leave the underlying OD-net as flat, unstructured and un-typed as possible. Unfortunately to make a model on a computer a certain degree of abstraction is required to approximate experience in binary form. Reconciling all of these issues is the challenge faced when implementing tools for Empirical Modelling and plastic applications.

From the survey and from personal experience there are 9 key problems identified with the *tkeden* approach to the representation of observables on a computer. These problems are split into two categories, names and types:

Naming

A1. Requirement for C syntax observable names There are times where the C syntax restrictions prevent giving an observable the name it really should have.

A2. Observable aliasing problems Having a single flat observable space is hugely problematic since each observable name must be unique. This is particularly problematic when combining models.

A3. Arbitrary choice of naming conventions Each modeller for each model is free to choose a naming convention. Such conventions are needed due to problems A1 and A2 above but are often difficult for others to understand.

Typing

B1. Observables hard to manipulate and search As the EDEN environment is unaware of certain relationships and structure there is little support for copying or

[4]If it looks like a duck, sounds like a duck and acts like a duck then it is a duck.

sophisticated search of observables in the flat space[5].

B2. Not scalable to larger complex models Difficulty with automating the construction of large models means that either tedious manual construction or inflexible automated construction is used.

B3. Primitive types inadequate Lists become unmanageably complex as a way of representing structures that are required.

B4. Ineffective reuse of existing models Components in models are not clearly separable and so it is difficult to reuse only parts of a model. This is also related to problem A2.

B5. Little support for shifting focus and contexts Switching between groups of observables depending on the current situation is difficult[6]. It is also not possible to move to a higher, more abstract focus where the individual observables are no longer important.

B6. No "embedded" interpretation Most of the interpretation of a model remains in the mind of the modeller and so is not easily transferred to others looking at the model[7].

All of these problems can be seen in and illustrated with existing models developed in EDEN. Problems A1, A2 and A3 can all be seen in Beynon's Lines model [Beynon, 1991] where observable naming has been a real difficulty. The examples in listings 3.1 and 3.2 show some of the observable names and definitions found in that model and it is clear that some syntactically restricted naming convention has been used to generate unique observable names for thousands of observables. What this convention is remains

[5] Using the DMT [Wong, 2003, p.181] it is possible to navigate the dependency graph to search for observables.

[6] Achieved by loading in other script files to make and undo bulk changes.

[7] The purpose of LSD is to describe protocols for interaction which gives the intended interpretation [Beynon, 1997c], however this is not used or well supported in EDEN.

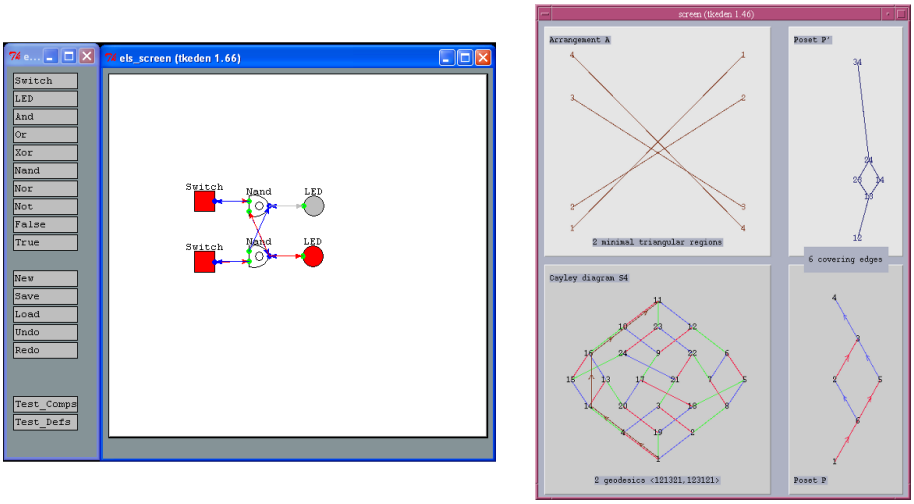

Figure 3.3: Left: Eden Logic Simulator. Right: Lines Model.

a mystery, even to Beynon at times[8], and is an example of problem B6 [Rungrattanaubol, 2002, p.189]. The syntax restrictions of A1 also go against the EUD guideline of making syntax errors hard (cf. 1 in table 2.2). The Lines model is not the only example of these problems as similar problems exist in almost all models of a significant size.

```
l1334
g2434
pbai_v
eqabbai_2
```

Listing 3.1: Lines model observable names

```
upbai_4 is check_int(if_assign(i_eq(pbai[4][4],UNDEF),
        one_int(1), one_int(0)),upbai_4_kind)
```

Listing 3.2: Lines model definition

[8]Although he would claim to be able to recover an understanding through interaction with the model [Rungrattanaubol, 2002, p.189], something which EM tools, such as EDEN, are particularly good at supporting.

To interpret the observables and definitions in 3.1 and 3.2 requires some detailed explanation of the convention used and even then having to manually search through thousands of observables to figure out what the model is doing is obviously extremely challenging. This relates to problems B1, B2 and B3. It should be noted that the definition given in listing 3.2 is actually part of the Eden translation from the ARCA notation so is not directly written by the modeller. The three problems, B1, B2 and B3, also feature prominently in the Timetable model [Beynon et al., 2000b] as well as many student coursework projects such as the ELS (Eden Logic Simulator) [Lee, 2007]. Both of these models contain large numbers of similar *components*, such as cells in the timetable or logic components in a circuit, and have attempted to automate the construction of these components using agents. In each case a different approach has been taken and they have proven to be very inflexible. Not only are they inflexible and complex but they also automatically generate thousands of observables following a particular convention and exacerbate the problems of A1, A2 and A3. The fact that they had to use different approaches for essentially the same problem is also an example of problem B4.

The *personal* focus of Empirical Modelling is partly why problem A3, that of personal naming conventions, has come about. Such conventions do not help in communicating an artefact to others, i.e. making an artefact *public*. The same *personal* focus leads to B1 due to an assumption that the individual modeller will have some mental model so knows what observables there are. Obviously such mental models remain in the mind and are not transferred to others when the artefact is. To some extent this relates to the EUD guideline of providing incremental disclosure (cf. 8 in table 2.2). Some solutions have been proposed by Rungrattanaubol, for example using dynamic annotations of scripts (described by other scripts) which helps to identify the meaning of observables (also helps with B6) [Rungrattanaubol, 2002, chapter 7].

Similarly the lack of structure and support for types (B3,B5 and B6) is a result of focusing on the *provisional* nature of a model rather than an *assured* artefact. It is open

to fluid change but cannot be solidified into specific identifiable entities. Problem B4 is an example of where the focus on *specific* (concrete) has led to models that cannot be made *generic* enough for reuse in other models. This has proven to be a significant limiting factor for EM. Many models end up reinventing the wheel with their code as modellers are unable to adapt models to their needs. Most models can only act as inspiration rather than a real resource to be used and a considerable number of students have made this point in the coursework reports. It is clear that this problem has its roots in many of the other problems which help to make such reuse difficult. Often it is this lack of modularity combined with B2 that students use as an argument for using Object-Oriented languages instead of EM.

The final point above highlights another aspect of the refinement process, that of non-linearity. Parts, or components, of a model may be refined at different rates with certain components becoming public, assured and generic whilst the rest of the model using those components is still personal, provisional and specific. This indicates a need to not only support plastic applications as a whole but plastic components of any size that may be combined in a hierarchical fashion to varying degrees of refinement (cf. B2 and B5). Again, looking back to EUD guidelines this problem of components matches with a need to support decomposable test units (cf. 7 in table 2.2).

The lack of support for context shifting (cf. B5) is often not mentioned explicitly in student models but does have an impact upon the choice of model constructed by students. For example, Beynon has attempted to develop models that require the kind of context switching abilities of B5. One such model involves a need to specify the car of a mother-in-law's husband where the car, the mother-in-law and husband could change. It is difficult to achieve this kind of switching without resorting to the crass approach of loading script files to make bulk changes. In the end these models are abandoned or adapted to avoid such problems.

3.3.2 Analogue and Process Dependencies

The concept of dependency is well developed in the EM conceptual framework. Since the focus in EM is on developing a rich model of state with which agents may interact to experiment, these dependency relationship concepts are passive in that they respond to change to maintain these relationships but do not themselves cause change. All change comes from agent interaction with this model of present state. Figure 2.4 highlights how dependency is only used to support the model of state and plays no direct part in describing processes and behaviours. Consequently the OD-net only gives a static representation of the current state of a model with the *experientially-mediated* associations [Beynon, 2011] being *latent* and only coming into action to deal with agent initiated change.

What is being missed by this is the potential for using dependency to describe processes[9], where there are *experientially-mediated* associations [Beynon, 2011] which are not about dealing indivisibly with change but are across time and so make change visible. Indivisibility is about being coherent in the presence of change so it is still important that there be coherence between and within these states[10]. The purpose of dependency in the EM conceptual framework is to enable experimentation (by maintaining the integrity of state) to take place in order to gain understanding and help develop an artefact. To this end EM has focused on providing a rich model of state that supports such experimentation (the OD-net). Agents, on the other hand, are there to do the experimenting by interacting and observing the artefact. What if, however, the artefact and experiment is about a process or involves associations across time? Often the most difficult systems to understand are dynamical systems and so computer support for developing dynamical artefacts that can be experimented with is important. Faraday's experiments to develop the electric motor involved a dynamic process and he used what Gooding calls *dimensional enhancement* when a "causal explanation is

[9] Processes should be included as *particular* things which can be observed [Smith, 1996, p.118].

[10] Changes in the current state are still propagated indivisibly and the change from one discrete state to the next is also indivisible to an observer.

sought" [Gooding, 2001]. Dimensional enhancement is the addition of a third and possibly fourth (time) dimension to a diagram or actual model, something which is done only when the simpler *state-as-experienced* has been grasped (through the use of *dimensional reduction*). The artefact itself is given a temporal component that can then be experimented with, making it a dynamical artefact. Today in chemistry, physics, biology and other fields it is becoming increasingly important to recognise the dynamic nature of the world around us.

The concept of coherent observable current state still plays a vital role even though the current state may be unstable. The view that there ever exists a stable concept of current state relates to the now debunked notion of natural systems being self-regulating and stable, that there is, for example, a "balance of nature". The role of dependency, it seems, is to rebalance the artefact into a stable state after each change has occurred. However, it is widely acknowledged by ecologists and others that "the natural environment... is in a constant state of flux" [Jacobs, 2007], and to abstract to a static conception of state is to ignore something vitally important. Brian Cantwell Smith also identifies a necessary move "away from treating the world in terms of static entities instantiating properties and standing in relation, and towards a view that is much more intrinsically dynamic and active" [Smith, 1996, p.36]. With Empirical Modelling's current focus on state such dynamical processes require the use of agents which can be argued goes beyond their remit of experimental interaction and observation. The artefact itself, the OD-net, should embody the dynamism found in the referent and become a *dynamical artefact*. Both ways of construing the world are valid, with agents or dynamical processes, and Empirical Modelling should allow for either approach to be taken depending upon the personal preference of the modeller.

Whilst the EDEN implementation of dependency has proven successful and is faithful to the EM concept of dependency, it does suffer from numerous problems that have come to light through years of modelling activity, and which draws attention to the need for a different conceptualisation of the role and nature of dependency. There

are 6 key problems identified by the survey and personal experience. These problems fall into two categories: conceptual (C) and technical (D):

Conceptual

C1. Feedback not supported Dynamical dependencies cannot be specified which prevents 2-way relationships and feedback situations from being easily described.

C2. Animation requires use of clock ticks All animation requires either procedural actions or dependence upon a clock observable.

C3. Unable to observe events Sometimes a change needs to be observed rather than just the value itself and this cannot be done with definitions.

Technical

D1. Not monitoring all dependencies Any observables used inside functions are not added as dependencies and so do not trigger updates.

D2. Contains procedural elements These elements allow for side-effects which causes concurrency and conceptual issues.

D3. Dynamic dependencies not maintained If the dependencies a definition describes change then the system does not track those new dependencies.

The first three conceptual problems are the ones which do need addressing, whereas the last three are only deficiencies in the implementation rather than a conceptual problem. Examples of C1, C2 and sometimes C3 have been encountered in many student projects. Going back to the digital logic simulator referred to earlier [Lee, 2007], the author of the model commented that he was unable to use dependency for the wires connecting components together because sometimes these connections were cyclic in nature. To resolve this he needed to resort to using the EDEN clock mechanism to tick through discrete instants and also maintain a concept of previous state. In this way the

next state was continually updated based upon previous state but all this was achieved using agency and so lost most of the benefits of using EDEN, lost the benefits of dependency. What was being attempted was the development of a dynamical artefact where a simple and static conception of state is inadequate. Another example is a model of the Water cycle and infrastructure in Singapore [Beng, 2006]. The water cycle is, as the name would suggest, cyclic in nature and again the author of the model commented on how dependency could be used everywhere except in one, randomly chosen, point because the cycle needed to be broken. Other such cyclic models include: Agent Based Bridges [UNK, 2005] and a Neural Networks Notation [Hammond, 2006]. Beynon has also played with this problem in the form of two blocks connected by a string [Beynon, 1989]. Pulling one of the blocks will move the other if the string is taught and vice-versa. Such dependency is 2-way and hence not possible in EDEN as it is cyclic.

Problem C1 seems to be the bigger issue, however, C2 is more common. Typically models need to be animated in some way and currently this involves some clock agent incrementing a tick observable that all other definitions depend upon. Whilst this works in many cases there is a clear need for a better conception of time. Animation is an example of an instability in current state and represents a form of dynamical system. Looking back to Gooding's work on Faraday, he describes how construals and physical artefacts were built which were changing without the intervention of the experimenter and how Faraday was observing these processes and influencing them through interactions [Gooding, 1990, p.149]. Such "animation" is not conceptually attributable to an agent, or at least it is not especially meaningful to think of it as such.

The final conceptual problem, C3, does not occur often in student projects but has been noted by Beynon in a train model where a conductor whistles. Observables are needed to know if the conductor is whistling or has whistled and this involves the observation of an event. Another example would perhaps be the observation of a button click by watching for both a press and release event from the mouse. Such observations would need a concept of previous state, however EDEN and the OD-net concept only

allow for the present state to exist in an artefact.

With regards to plastic applications it is important to allow for dynamical systems, but also by moving away from agency towards dependency it is hopefully easier to generate a more *assured* and formal description. Dependency is functional in character, as opposed to the imperative nature of agency. As a consequence it reduces the difficulties of orchestrating side-effects (cf. Coherence in §2.1.3) and can draw on the mathematics of functions (and data-flow systems [Wadge and Ashcroft, 1985]). EDEN as it stands, and EM in general, requires an excessive use of agency which is detrimental to this goal of becoming *assured*[11].

From a technical point of view problems D1 and D2 can be resolved by following particular conventions or using special features within EDEN. For D1 simply pass all required dependencies as parameters to the function or if this is not possible then use a special feature that allows dependencies to be manually added. Ultimately this should not be necessary. For D2 the modeller can avoid using side-effects inside functions used by definitions. More difficult to resolve but also quite uncommon is D3. Ashley Ward has discussed this in his thesis [Ward, 2004] and at one time decided to deprecate the ability to dynamically change the dependencies within a definition, however, this broke some models. The problem lies in the ability to dynamically generate observable names from strings inside a definition. If these observable names change for some reason then EDEN does not keep track of the new dependencies for those new observables.

3.3.3 Notations and Agents

In the EM framework, agents interact with, observe and construct the artefact. The artefact is represented as an unstructured OD-net. To help with this, task-specific *definitive notations* are used to provide appropriate structures and operations for agents to use with the OD-net. Task-specific notations[12] are nothing new and have been

[11] Especially in concurrent systems where concurrent dependency maintenance is a far easier problem than concurrent agency.

[12] Originally the term *notation* was used in EM to mean an incomplete (non-Turing complete) language. There is no need for a specialised language to be complete. The term should also include visual notations

explored a great deal by the EUD community. The benefits are obvious, the end-user may work with a notation that relates to the task at hand and their domain of interest, rather than being more generic but complex[13]. One of the EUD guidelines is to use domain-oriented languages where possible (cf. item 4 in table 2.2). However, there is an important difference in motives between the EUD use of task-specific languages and the use of definitive notations: EM is trying to avoid being grounded in a foundational way in any particular representation. EUD, on the other hand, seems only concerned with usability rather than foundational issues. Concerns about foundations are discussed in depth by Smith [Smith, 1996, p.81]. No single approach to representation can do justice to the extraordinary diversity of things (in experience) that need to be represented.

It is, however, well acknowledged that task-specific languages suffer from a collection of problems: the difficulty of creating many different languages, inconsistencies between languages and knowing what any particular language should and should not contain [Nardi, 1993, p.50]. Each of these problems can be found in the survey of EDEN. The issues identified by Nardi are further exacerbated by the nature of the EM refinement process. Initially all interaction needs to remain free from unwanted restrictions that specific notations will tend to cause. To be free and yet use definitive notations may require unrealistic notational flexibility. At the other end of the process these notations may need to be refined to become far more specific. Nardi's third problem of deciding exactly what needs to be in a particular notation becomes a part of the refinement process of an EM artefact and so may be specific to a particular artefact, meaning that many different unique notations are required which is "expensive". Without this notational flexibility the artefact will be unduly constrained by the existing notations.

A partial (and perhaps unsatisfactory) solution to this, as suggested in Repenning and Ioannidou's guidelines (cf. §2.1.4), is to have a meta-domain language to connect

as well as textual.

[13]Turing tar-pit: "Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy" [Perlis, 1982].

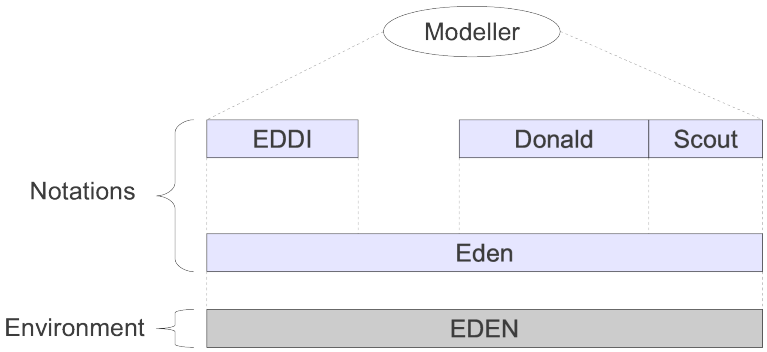

Figure 3.4: Notation layering in EDEN

the other languages together and allow for generic situations where a specific language does not exist. In EDEN this is the Eden language. The Eden language is in fact a generic representation of the OD-net so it is the OD-net that fundamentally provides the meta-domain link between notations. These problems and others have been identified in EDEN by the survey:

Notations

E1. Inter-notation communication difficult All notations translate in a unique way to the underlying Eden notation and so users must understand all these conventions. Connections also have to take place in the domain neutral Eden notation.

E2. Inconsistencies between different notations Major syntactic differences exist between each notation and this can be confusing, especially if large numbers of domain specific notations are eventually supported.

E3. Custom notations for each different domain It is not practical to have a custom notation for each domain of interest and so having to fit with one of the existing specialised notations or the entirely flat world of Eden is a major cause of frustration.

E4. Focus on textual notations only The notation mechanism involves translating

DoNaLD	Eden Definition	Eden Value

within desk {
line E
point NE, SE
E = [NE, SE]
}
	_desk_E is
	line(_desk_NE,
	_desk_SE);
		['L',['C',365,465],
		['C',365,115]]

Table 3.1: Illustrating DoNaLD to Eden translation

to the Eden notation and so a textual approach is encouraged. This discards the possibility of a visual notation and is less direct and interactive than desired.

E5. Notations only work for modeller and not other agents All agents have to be written in the domain neutral Eden notation and cannot take advantage of types, structures and operators available in other notations. Related to problem E1.

The first problem (E1) is best shown by an example of how definitive notations translate down to the underlying Eden notation. Table 3.1 shows how a simple DoNaLD definition of a line is converted into Eden and what the resulting value of the observable is. This example shows how the list type in Eden has to be used to represent the concept of a line and how this could be difficult for end-users to interpret. The problem gets far worse for more complex types such as Scout windows (cf. listing 3.3), largely because these representations are not *self-describing*[14] once translated (cf. listing 3.4).

With many of the existing definitive notations, including EDDI, Sasami, DoNaLD and Scout, there are object-like concepts that require translation into Eden list structures. The prevalence of such object concepts would indicate a need for the underlying and integrating notation to better support such concepts. In other words, the OD-net

[14] A term often used with semi-structured representations, such as XML, where structural descriptions are included with the content because generic descriptions do not exist.

```
%scout
window A1 = {
    type: TEXT
    frame: ([{A1_X1, A1_Y1}, {A1_X1+gridsquare_width.c,
            A1_Y1+gridsquare_height.r}])
    font: A1_font
    bgcolor: A1_bgcolour
    fgcolor: A1_fgcolour
    bdcolor: A1_bdcolour
    border: A1_border
    relief: A1_relief
    alignment: CENTRE
    sensitive: ON+ENTER+LEAVE
    string: a1
};
```

Listing 3.3: SCOUT window example

```
A1 is [0, [formbox([A1_X1,A1_Y1],[A1_X1 +
    column(gridsquare_width), A1_Y1 +
    row(gridsquare_height)])], a1,[0,0,100,100], "pict1",
    DFxmin, DFymin, DFxmax, DFymax, A1_bgcolour, A1_fgcolour,
    A1_border, 3 ,1.0 + 4.0 + 8.0, A1_bdcolour, A1_font,
    A1_relief, "A1"];
```

Listing 3.4: SCOUT window translated to Eden

needs to support these kinds of structures to make inter-notation communication a little less daunting.

The second most contentious issue is E4 where definitive scripts are far too static to deal with the dynamic nature of artefacts. This is strongly connected with the previous argument in §3.3.2 and the motivations behind the creation of Subtext (cf. §2.1.3) which is trying to move away from paper-centric approaches to programming. If dynamical systems are to be supported then the foundational role of definitive scripts is no longer practical since a fixed textual representation of state cannot do justice to the dynamic nature of the artefact[15]. Even without the introduction of dynamic concepts a script is not as interactive and observable as the system actually is internally, and could also be in violation of the EUD principle of directness since scripts rely on indirect textual manipulation of an entity.

Problem E3 has already been discussed and is a well known problem. E5 illustrates a technical deficiency in the way in which EDEN has chosen to implement automated agency. Instead of allowing agents to act above and outside of the artefact and its corresponding notations, EDEN has implemented agents directly in the "foundational" language Eden which means they are forced to interact and observe the OD-net directly with no alternative views or interpretations being possible[16]. Agency needs to be fundamentally removed from the artefact itself and allowed to operate at an entirely different level. Automated agents need to be given the same role as the modeller. There is much more to say on the nature of agents and how they should be implemented, but for this thesis the focus will stay with the OD-net. The observation made by E5, with regards to agency, is all that will be said since it does impact upon the nature of the OD-net.

[15] This is not to say that task-specific notations are not important, they still have a role to play for agent refinement.

[16] It is possible for Eden agents to use an *execute* command to run code written in other notations, however, this has proven to be exceptionally complex to write and subsequently understand as it involves manipulating strings of scripts.

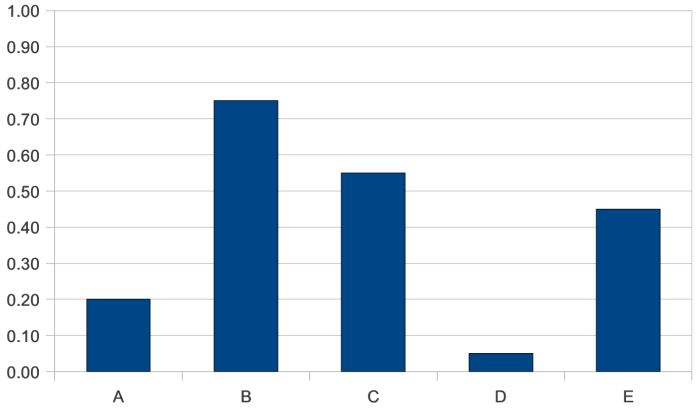

Figure 3.5: Fraction of student models that explicitly or implicitly mentioned problems in each category

3.3.4 Summary of Survey

This section has identified a total of 20 problems with both EM concepts and their current tool EDEN. The problems have been listed together in table 3.2. These problems form the basis for the questions asked by this work and in the next section of this chapter solutions are proposed, followed by specific approaches to be taken by the rest of this thesis.

As a part of the analysis involved in discovering these problems, 20 Web-EM coursework papers were reviewed to see how often each problem came up. The graph in figure 3.5 shows the results grouped into the categories of problems identified. Figure 3.6 breaks this down further into the individual problems. This analysis is limited in that most students are directed to explore particular kinds of models that avoid certain technical issues and often they do not report problems they had in the written paper. Also, these are individual and personal projects so do not identify issues with scaling up, which is where my additional EM experience and that of others has also played a part in identifying the problems. Despite this the analysis clearly shows what the most common

Table 3.2: 20 problems of EM and its tools

A1	Requirement for C syntax observable names
A2	Observable aliasing problems
A3	Arbitrary choice of naming conventions
B1	Observables hard to manipulate and search
B2	Not scalable to larger complex models
B3	Primitive types inadequate
B4	Ineffective reuse of existing models
B5	No support for shifting focus and contexts
B6	No embedded interpretation
C1	Feedback not supported
C2	Animation requires use of clock ticks
C3	Unable to observe events
D1	Not monitoring all dependencies
D2	Contains procedural elements
D3	Dynamic dependencies not maintained
E1	Inter-notation communication difficult
E2	Inconsistencies between different notations
E3	Custom notations for each different domain
E4	Focus on textual notations only
E5	Notations only work for modeller and not other agents

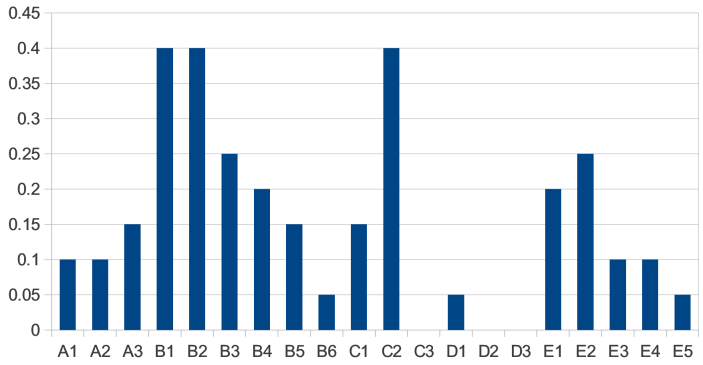

Figure 3.6: Fraction of student models that explicitly or implicitly mentioned specific problems

problems are (of those included in the non-exhaustive list in table 3.2). It seems that type issues such as the need for objects and some need for active forms of dependency are most common and so will be focused upon.

3.4 Looking for Solutions

For most of the issues identified with EDEN and the EM concepts there are suggestions of solutions to be found in other research areas and industry. Chapter 2 introduced End-User Development and three of its tools, along with other XML related technologies. These approaches are to be explored here as possible ways of resolving many of the technical and perhaps conceptual problems with EDEN and EM. For now the conceptual consequences of using these practical solutions will mostly be put aside to be revisited later (chapters 6 and 7), but since one of the founding objectives of EM is to develop "theoretical frameworks that do justice to [the] practice" [Smith, 1987] there is a need to understand how these EUD and industry technologies fit within Empirical Modelling and how they can be appropriately and beneficially moulded into its framework. Due to needing to constrain this work the issues relating to agency and the refinement of

their interactions will not be focussed upon at present, leaving the ideas behind the LSD notation as an appropriate solution for the time being. The focus instead will be on the OD-net itself and how to better support the notion of *context* and *construal* on a computer (cf. figure 3.2) with regards to the ability to transition to the more refined plastic applications level.

3.4.1 Richer Types and Semi-Structure

Many of the problems identified in §3.3.1 on naming and typing of observables are related to the motivations behind the development of structured and semi-structured representations [Buneman, 1997], including object-oriented languages. These problems highlight the need for giving groups of observables a collective identity and for developing structures and categories to enable interpretations, manipulations, communication and reasoning. The motivations for this do not need to be stated as they are well known from object-oriented programming, mathematics, the arts and philosophy, in other words practice shows it is important. What is problematic is the lack on an ontological theory and that "traditional ontological categories ... are both too brittle and too restrictive" [Smith, 1996, p.45] as well as being overly committing. As Brian Cantwell Smith puts it in "On The Origin of Objects", we want:

> *"notions of objects that are fluid, dynamic, negotiated, ambiguous and context-dependent ... rather than the black-and-white models inherited from logic and model-theory."* [Smith, 1996, p.46]

Plastic applications and Empirical Modelling are striving to allow for exceptional flexibility which requires exceptional representational flexibility. It is argued, correctly, in EM that preconceiving structures and classifications for observables is not possible in an exploratory, experimental and experiential process that starts life as a personal, subjective, provisional and specific construal. To be provisional and subjective means that fluid representations are vital. To this end the issue of structure has largely been

avoided, with EDEN's flat observable space being an example of this. Instead *definitive notations* are used to provide task-specific views which includes the grouping and classification of observables that relate to the task at hand but that in principle is intended as only one view of the otherwise unstructured OD-net.

Unfortunately it is clear from §3.3.1 and §3.3.3 that the use of notations to provide representations is inadequate and that richer but fluid representations need to exist within the OD-net itself[17]. The ontological issue cannot be ignored in the foundations of Empirical Modelling, especially if the transition to a program is being sought.

Interestingly there has been some suggestion of enriching the OD-net with object-oriented characteristics before. Edward Yung, the original developer of EDEN, suggested in the future research section of his Masters thesis on EDEN that object-oriented concepts should be considered to allow for structured data types and inheritance [Yung, 1990, p.101]. Similarly, Allan Wong proposed ways of organising definitions into *containers* and proceeded to develop new tools[18] and interfaces based upon this idea [Wong, 2003, p.167]. What these comments and attempts fail to fully appreciate is the inflexibility of traditional OO approaches and how significant and fundamental the issue of representation really is.

The answer may lie in the ideas of semi-structured data, as already indicated, and those of prototype-based languages. In both of these approaches a degree of flexibility is possible and accepted as necessary. Subtext (cf. §2.1.3) takes advantage of a tree structure, as does XML. However, the most unrestricted form of structure is perhaps the edge-labelled directed graph which can be related to the most basic algebraic structural concept, that of a *magma* [Rosenfeld, 1968, p.90][19]. Without any additional object-oriented concepts such as message passing, or any additional complexity as present in

[17] Especially since some inflexible representations in the form of types are already in the OD-net causing difficulties.

[18] WING (WINdowing and Graphics tool) [Wong, 1998] and EME (Empirical Modelling Environment) [Wong, 2001] which formed the basis for the DMT [Wong, 2003, p.181].

[19] A magma may also be known as a *groupoid* and contains a single set and a single binary operation closed over that set.

XML descriptions, the idea of using a pure graph to provide structure is perhaps a viable approach for EM and plastic applications as a way of structuring state. The idea being to semi-structure the OD-net in an attempt to resolve the issues identified previously without becoming too inflexible in the presence of change. The concept of *schema* could then be taken from XML and research on semi-structure as a means of developing refinements on interpretations for agents as the artefact moves from construal to a plastic application. Schema may be a new way of interpreting *definitive-notations*. Capability-based security[20] may also be applied to the graph as a way of restricting agent actions and observations through privileges.

The idea of a semi-structured OD-net is rich with possibilities. Some possible consequences are that observables could be manipulated using cloning of sub-graphs, that richer hierarchical types can be supported, that components can be isolated and refined at different rates. There are even greater consequences that will become apparent through this thesis[21]. The main argument against this is that it may still not be fluid and dynamic enough in that even with a semi-structured approach there is some need to commit fairly early to particular representations. Albeit considerably later than fully structured approaches and with a greater possibility for subsequent re-factoring if it proved to be inappropriate. How beneficial a semi-structured OD-net would be, and how problematic, is unknown.

3.4.2 Taking Advantage of Dependency

A need to support dynamical artefacts was identified in §3.3.2. The basic problem is that cycles over time do exist and that some processes are not appropriately described using agents. There is a clear need to consider artefacts which are not simply static entities. A fundamental aspect of EM is to enable experimental interaction and observation of an

[20] Capability-based security is an alternative to Access-Control-Lists in operating systems [Levy, 1984; Miller et al., 2003]. It involves particular users and processes (or agents in our case) being given tokens to act not only as object identifiers (or node references in a graph) but also to say what permissions they have. This is a decentralised approach to security that has been explored in the CapROS operating system (successor to EROS).

[21] Computation as graph navigation being the main one.

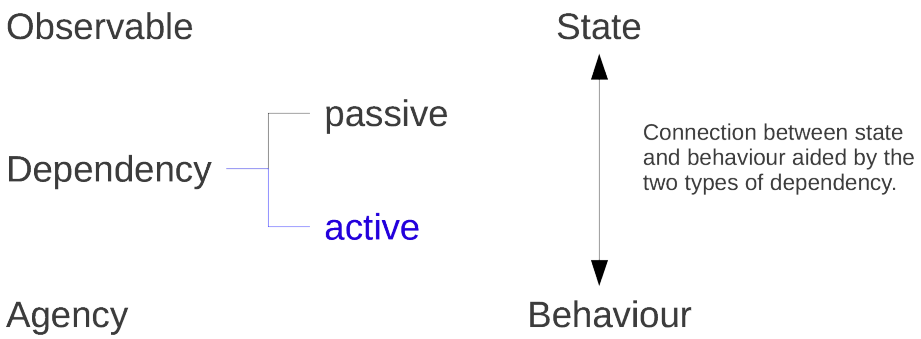

Figure 3.7: ODA with active dependency.

artefact in order to evolve it. The goal of plastic applications is to evolve it sufficiently far that it becomes like a program. One way to enable experimentation with an artefact is to use explicit dependency to maintain a coherent state in the presence of change, this has been the EM approach to date. So to enable process-like artefacts and animation, is it possible to also use dependency? Can dependency be used to not only support rich models of state but to also support processes (cf. figure 3.7)?

The advantage of the EM notion of dependency is its indivisible nature that prevents observation and interaction by agents from occurring before any previous change has been fully propagated. This enables agents to observe only coherent state rather than incoherent state that is still being computed[22]. This characteristic of indivisibility must be maintained with dynamical artefacts. To retain this indivisibility there must remain a notion of coherent current state, somewhat like a snapshot that agents may observe. This snapshot notion is in fact well aligned to scientific experiment where the experiments typically take such snapshots at discrete points in time for later analysis[23]. How frequently such snapshots can be taken depends on the complexity and sophistication of the experimental apparatus, which in the case of a computer artefact is very frequently indeed. So indivisibility must exist within a snapshot and between snapshots.

[22] In traditional imperative programs it is difficult to identify such globally coherent state so meaningful observation and experiment are all but impossible.

[23] As should be obvious, experiments are often left unassisted between such snapshots which is why a notion of unassisted process is needed for EM artefacts.

Dynamical systems are described by evolution functions which give a trajectory through time to a particular property [Weisstein]. These systems can be discrete and if multiple properties are involved they can be synchronised, which gives a degree of coherence. Such evolution functions could in fact be described as dependency definitions by allowing for dependencies on previous values of observables. If appropriately implemented this would enable indivisibility between states and, with a special case where a definition only depends upon present values, within individual snapshots of current state as well. The result is a discrete set of snapshots that record the history of an artefact and, through active-dependencies, describe a trajectory into the future.

Back in chapter 2 the EUD environment Forms/3 was introduced (cf. §2.1.3). Forms/3 implemented a similar concept for spreadsheet style formulas and cells where each cell has a temporal vector to store its history. The formulas can then refer to previous values of a cell and this enables animation. The concept works well and relates well to the proposal here. The challenge then is to integrate this notion of active-dependency with the now semi-structured OD-net to provide a dynamical artefact that can be refined in an EM manner. The additional benefits of this regarding refinement to a plastic application have already been covered in §3.3.2.

3.5 A New Tool?

What the proposed solutions in §3.4 describe is a dynamical[24] semi-structured OD-net that supports the Empirical Modelling process. There is no existing interactive tool that supports this concept and the existing EM tool EDEN cannot be adapted this radically (cf. §7.1). Therefore, a new prototype will need to be constructed to test out the ideas. There are four key questions to be answered in the following chapters of this thesis:

1. Is it possible to implement a dynamical semi-structured OD-net as an EM tool?

[24] Dynamical is used as opposed to dynamic because it indicates that the system changes itself rather than that it is capable of being changed.

2. Does the idea actually resolve the problems identified with EM for plastic applications?

3. What is now possible and does this help move EM towards its notion of program?

4. Are there any conceptual consequences of making these changes and how can they be dealt with?

What has not been mentioned, and has been deliberately left out, is the role and nature of agency. It is not the concern of this prototype to resolve the issues with agency and so it is free to choose any implementation. One thing EM tools have suffered from is the lack of modularity and lack of integration with existing technologies. Dependency-injection, which traditionally uses XML, would seem to fit well with the new semi-structured approach and so could make use of the semi-structured OD-net as *glue* for external components, which can be conceived of as agents in the new tool (Cadence). In this way existing technologies of almost any kind could be flexibly linked with the OD-net. This is perhaps one, less radical, means of bringing EM and plastic application principles to existing programs and gives a practical framework with which to enable end-user development of applications using EM principles. Although developing applications in a radically EM and plastic way from the start is the aim, this allows for a more incremental transition away from traditional approaches which may enable EM ideas to be accepted by the software industry.

Chapter 4

Cadence: A Prototype Tool

Having identified some key issues with EM and EDEN in chapter 3, and found possible solutions, the next step is to introduce a new prototype tool that has been developed to go some way towards finding out whether these solutions work in practice. The prototype tool has been called *Cadence* and constitutes a major contribution to this thesis. It is one attempt at developing the dynamical semi-structured OD-net described previously. Cadence is envisioned as a future alternative to EDEN which resolves many of its issues and better enables Empirical Modelling concepts and principles to support plastic applications, as well as generally showing how EM can be useful for software as a whole. This chapter will introduce Cadence and identify ways in which to implement a semi-structured OD-net and how to make it dynamical. The implementation architecture is then given, which includes discussion of user interfaces and other extensions developed with a C++ API. Subsequent chapters will illustrate the use of Cadence with example models and explore conceptual aspects, before finally relating it all back to Empirical Modelling and concluding with how it helps support plastic applications.

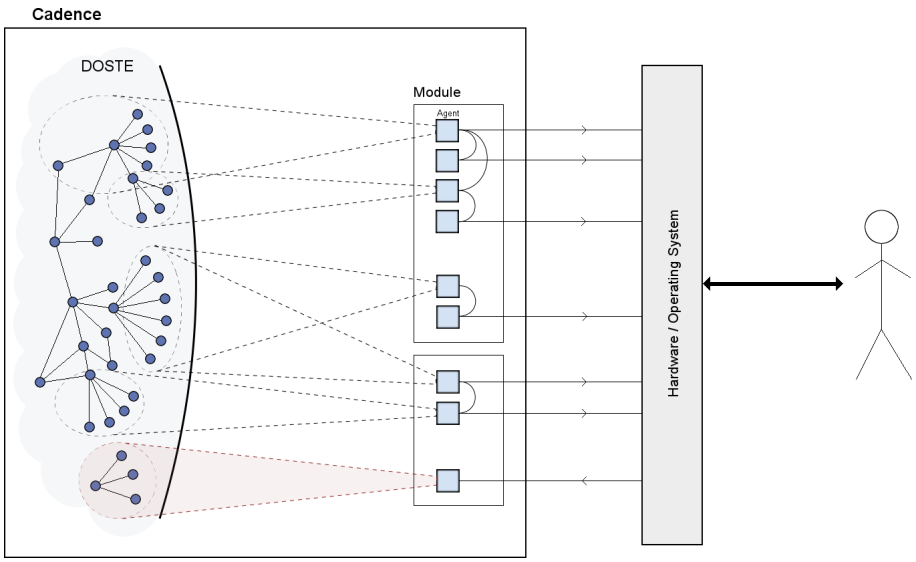

Figure 4.1: Cadence conceptual model

4.1 What is Cadence?

The name Cadence has been chosen to emphasise a creative flow of software development, not dissimilar to the creation of music or poetry, and because of its dynamical nature. Cadence is a software environment that supports the development of construals and a refinement of those construals towards programs by embodying the principles of end-user development (cf. §2.1.1) and Empirical Modelling (cf. §2.2.2). It can be thought of as an operating system[1], as a virtual machine, as a modelling tool; it is all three of these, the distinction between them being somewhat inconsequential. In other words, it is intended to be a plastic software environment.

The tool implements the idea of a dynamical semi-structured observable-dependency network, as described in §3.4, along with the modular glue interpretation for agents. There are, therefore, two key parts to Cadence which are shown in figure 4.1: the OD-net core called *DOSTE*[2] and the agents which interact with it. Agents are provided in a

[1] A version of Cadence did boot as an independent operating system on 32-bit and 64-bit multi-core systems. This version was not maintained.

[2] Definitive Object State Transition Engine, but this is for historical reasons [Pope, 2007].

modular fashion[3] and may interact with or observe the OD-net. This interaction is live, immediate and unrestricted so that construals at the personal, provisional end of the spectrum are supported. Agents may act independently or, more often, act as mediators between the OD-net and any human users via a computer's hardware and peripherals, as depicted in figure 4.1. Agents provide means of observation by interpreting the OD-net in specific ways, and also provide means of interaction through controlled modifications to the OD-net. Refinement may take place by placing more restricted agents between the user and the artefact. The focus of this thesis is on the Cadence OD-net and not so much on the agency currently supported.

4.2 Semi-structuring the OD-net

The problems identified in §3.3.1, and to some extent those of §3.3.3, show a need to group observables into either name spaces or structures of some kind, albeit flexible ones. A solution was proposed in §3.4.1 which suggested that the OD-net should be semi-structured in a graph-like manner and that there should not be classes but that sub-graphs could be cloned instead to make new structures. The core of Cadence, DOSTE, implements the OD-net as an edge-labelled directed graph in its simplest form: a *magma* (cf. §3.4.1). This section describes how Cadence has achieved this semi-structuring.

4.2.1 How to Introduce Structure

Since DOSTE is to implement structure as a graph, graph terminology will be used. However, it is helpful to relate graph structures to other concepts such as observables and objects. Figure 4.2 shows different terminology for simple graph structures. A node corresponds to some entity, object, *observational context* or value, depending on choice of terminology and the current interpretation being given to a node, and is identified by an Object-IDentifier (OID). An edge from a node represents some named property, attribute or component of it. The edges are also identified (labelled) with OIDs. Some

[3] Agents are currently C++ classes but this is far from the desired way of implementing agents.

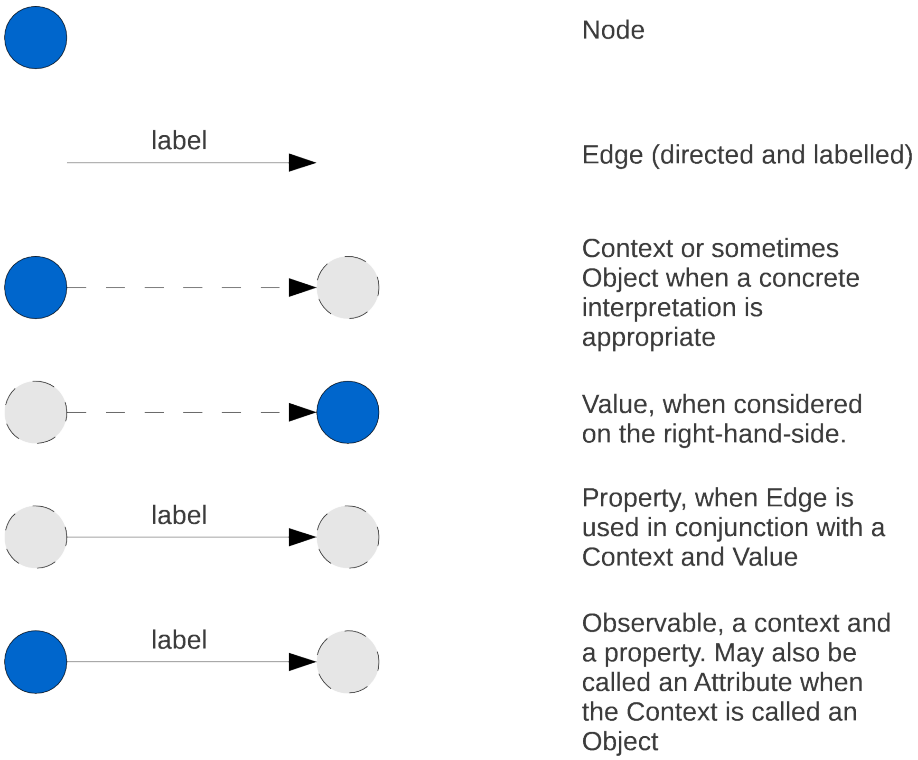

Figure 4.2: Cadence terminology for graph structures. Dark (blue) nodes and solid lines are what the terms refer to, whilst the grey and dashed lines set the context for use of those terms.

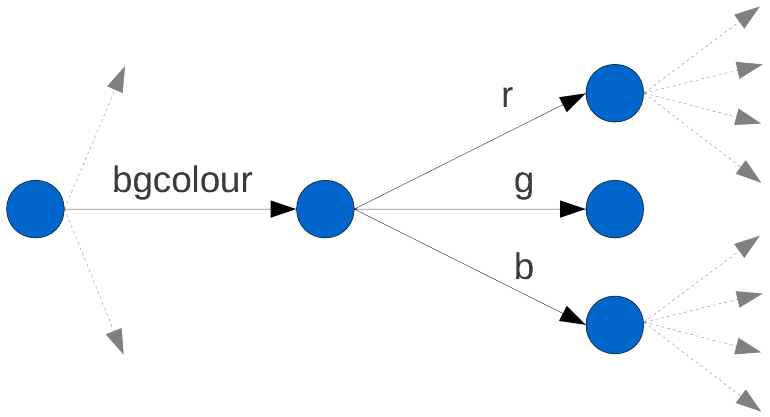

Figure 4.3: A graph example of representing a colour

OIDs can be mapped, by agent interpretation, to strings and numbers so that edges appear to have names, the same can be true for nodes. Due to edges and nodes both being identified by OIDs, it is possible to use a node OID to identify or select an edge - an important characteristic to be revisited (cf. §4.3.1). An observable is a node-edge pair which is connected to, or points to, another node that would be the *value* of that observable. The node part of an observable may be considered as a form of *observational context*, with an observable's value node either being interpreted as a plain value or an object. An observational context is typically the root focus of the current observation[4] and structures within this (i.e. the values of observables) are usually referred to as objects. Edges of *observational contexts* are observables whilst edges from value/object nodes are often referred to as attributes.

There is a need for some nodes to be interpreted in a special way to allow for numbers, but conceptually a number node is no different from any other. The graph in figure 4.3 shows an example of an observable called 'bgcolour', inside an observational context, which has three attributes for each colour component. The values of each attribute are nodes which correspond to some integer. Another interpretation, when the observational context is shifted to the colour object, is that 'r', 'g' and 'b' are observables

[4]The notion of "current observation" is fluid so what is deemed an object in one moment may be considered as an observational context in the next.

and not attributes. This shows how terminology shifts as the observation changes focus.

4.2.2 Developing a Textual Notation

A basic textual notation has been developed as a part of Cadence to enable a user to interact with the underlying DOSTE graph and as a way of describing the graph in text such as this thesis. The notation is called DASM which is short for DOSTE Assembly since it is best thought of as a form of assembly language with the potential for higher-level notations, including visual ones, to be placed above it. Originally it was compiled into a form of byte-code to be loaded by the operating system version of Cadence, but is now directly interpreted. At present it remains the only notation with which to interact with Cadence other than directly interfacing via C++ or one of the interface extensions discussed in §4.4.5. As a consequence all models and examples are based upon DASM so a good understanding is important to fully appreciate the work of subsequent chapters.

The DASM notation is unlike other semi-structured data approaches such as XML. Instead of representing static data (as a tree in XML's case), DASM represents state which is intended to be changed interactively. Individual DASM statements can be entered into Cadence (via some interface) to build up the artefact inside DOSTE[5], so in this way DASM should be considered as an interactive interface rather than a script. There is some similarity to Subtext, only DASM is a textual representation of what Subtext is trying to do more directly. Both are attempting to allow live manipulation of a software artefact's internal representation.

One of the simplest statements in DASM is a query that navigates the graph and returns the node that is reached. This involves specifying a start node and then giving the edges to follow. Nodes and edges are all labelled and these labels can be single words or numbers but may also be more abstract identifiers. There is a special reserved word called '**this**' which is the root node in the current context[6]. '**this**' may

[5] There is some similarity between DOSTE and the JavaScript Document Object Model (DOM), which XML in the form of HTML helps to build.

[6] *current context* can be changed to focus on another area of the graph and is a property of the interface through which the DASM script is being entered. It is possible to have unconnected graphs

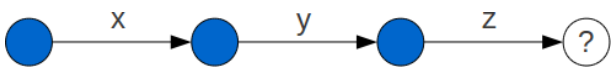

Figure 4.4: Navigating before an assignment

also be written as a *dot*. The example in listing 4.1 (cf. figure 4.4) starts from the '**this**' node and navigates along the edges 'x' then 'y' then 'z' to reach an unknown node. If the system has never been told what node should be pointed to by these edges then it will always return the '**null**' node where every edge that leaves the '**null**' node points back to the '**null**' node[7].

```
this x y z
```

Listing 4.1: DASM graph query

Following from this is a simple form of definition, shown in listing 4.2, where the node an edge points to can be changed to some fixed node (cf. figure 4.5). Instead of returning an unknown node a new node is given and the node the last edge originates from is returned. Having the origin node returned allows several assignments to be chained together, which can be seen in listing 4.3.

```
this x y z = 50
```

Listing 4.2: DASM edge assignment

The first example in listing 4.1 would now return the node '50' instead of '**null**'. It is important at this stage to think of '50' as being a representation of a node and not a numeral that represents a number. It is simply being used as an identifier and for it to be interpreted as representing a number requires other structures to be in place for numerical operations. These structures and relationships will be introduced later but for

in DOSTE with several different users using different interfaces with different contexts so that they can work independently.

[7] In this way there is never a problem with an edge being undefined.

```
this x y
    z = 50
    a = 4
    b = 5
```

Listing 4.3: DASM chained assignments

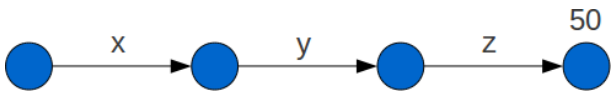

Figure 4.5: Navigation after an assignment

the moment assume the system has no knowledge of them.

To make an edge point to a node from another part of the graph, a path can be given on the right-hand-side (rhs) of the equals (cf. listing 4.4). It is critically important, however, that this be surrounded by brackets as otherwise the first identifier will be interpreted as the node the edge should point to instead of navigating the whole path. The remainder of the path would then act as a query to be combined with the left-hand-side (lhs) and will return a result, as demonstrated in listing 4.6 which shows how the example in listing 4.5 would be interpreted.

```
this x y z = (this z y x)
```

Listing 4.4: DASM assignment using a path

With what has been shown here it is possible to build simple structures and make basic queries by giving a path through the graph. Another useful feature worth introducing here is that of *context variables*. These are place holders for storing the result of some query for use elsewhere in the script which saves having to perform that query everywhere[8]. All context variables begin with the '@' symbol as shown in listing

[8]Context variables substitute in their current value when used so if the variable is subsequently changed any scripts that used it previously are not changed.

```
this x y z = this z y x
```

Listing 4.5: Incorrect assignment in DASM

```
this x y z = this;
this x y z y x
```

Listing 4.6: Interpreting an incorrect DASM assignment

4.7.

```
@xy = (this x y);
@xy z = 50
```

Listing 4.7: DASM context variables

The example in listing 4.7 is equivalent to the first assignment example in listing 4.2 but now every occurrence of '**this** x y' can be replaced by '@xy'. These context variables are an alternative to using the current context '**this**'. The current context could change, and often does, so context variables provide a stable means of referring to certain nodes.

When constructing a graph it is also necessary to refer to new nodes for edges to point to that can then themselves be filled with edges[9]. For this purpose there exists a '**new**' keyword which returns a unique and unused node. It can be placed at the beginning of any statement as a start node for graph navigation (cf. listing 4.8). If used on the right-hand-side of an equals it must always be contained within brackets otherwise it will be interpreted as a *label* (OID) rather than a keyword.

Both of the examples in listings 4.8 and 4.9 give the same result, although the second is shorter and more common in examples. With these constructs it is now possible

[9] Conceptually all nodes already exist, as do all edges. Unused nodes have all their edges pointing to the 'null' node.

```
this button = (new);
this button
    x = 10
    y = 10
    caption = "Button";
```

Listing 4.8: Node construction approach 1

```
this button = (new
    x = 10
    y = 10
    caption = "Button"
);
```

Listing 4.9: Node construction approach 2

to develop any graph structure supported by DOSTE. So far the notation shows the richer nature of observables and observational contexts than those found in the existing EDEN tool.

4.2.3 Cloning Sub-graphs

One of the benefits of having structured observables and contexts is the ability to manipulate these structures, a key problem with EDEN with regards to scalability. Cloning of structures is an example of this and is a good way of replicating a large or complex collection of observables, including their definitions. There is a large body of research and practical examples of cloning which can include some quite complex inheritance mechanisms, with Self providing inspiration for this work (cf. §2.1.3), along with Subtext and JavaScript. In Cadence cloning is a direct and complete copy with no in-built notion of inheritance. The DASM notation provides a special keyword called '**union**' that performs a copy of the object on the rhs into the object on the lhs. If there are two

edges with the same label then the one in the object on the right will replace the one on the left. The operation will return the left hand object.

```
.test1 = (new
    a = 0
    b = 4
);
.test2 = (new union (.test1)
    a = 3
    c = 6
);
```

Listing 4.10: Cloning sub-graphs

The example in listing 4.10 first creates a node with edges 'a' and 'b'. The second part then makes a node which first copies all the edges in the first and then modifies and adds to them. In the 'test2' node there are 3 edges: 'a', 'b' and 'c' with values 3, 4 and 6 respectively. Another example, in listing 4.11, shows the use of multiple '**union**' operations to create a moveable window from existing prototypes.

```
this mywindow = (new
    union (@prototypes window)
    union (@prototypes dragable)
    title = "Test Window"
    width = 200
    height = 100
)
```

Listing 4.11: Combining graphs with union

As the system is a graph, and may potentially be cyclic, there is a problem with the cloning mechanism. By default the '**union**' operation performs only a shallow clone where edges are copied but still point to the same value object. In some cases it is

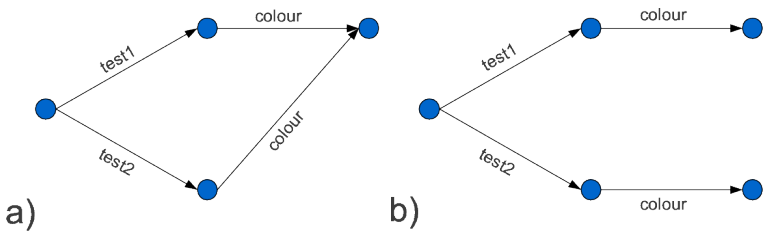

Figure 4.6: Results of shallow (a) and deep (b) cloning.

desirable to perform a deeper clone that will also clone a value node as well and use this as the value for the copied edge. DASM provides an annotation called '%deep' which flags a particular edge as needing a deep clone when the '**union**' operation is performed on it. Included in listing 4.12 is an example that would require deep cloning and the subsequent example in listing 4.13 shows how this can be achieved with the annotation. In the first case, if the colour values are changed they would change in both 'test1' and 'test2' due to the colour object being the same. The second example fixes this problem with the resulting graph shown in figure 4.6.

```
.test1 = (new
    colour = (new r=0.0 g=1.0 b=0.0)
);
.test2 = (new union (.test1));
```

Listing 4.12: Shallow clone example

```
.test1 = (new
    %deep colour = (new r=0.0 g=1.0 b=0.0)
);
.test2 = (new union (.test1));
```

Listing 4.13: Deep clone example

4.3 Making the OD-net Dynamic and Dynamical

In order for the DOSTE graph to be an OD-net it must support the current EM concept of dependency (cf. §2.2.2). Dependency allows the observable graph structure discussed previously to update coherently in the presence of external change; it makes the OD-net dynamic. In §3.3.2 limitations were identified with only using dependency in a passive sense and relying on agency for all change. A solution was proposed in §3.4.2 to make the OD-net support dynamical systems by allowing dependencies to exist across time. The proposal is to make the OD-net not only dynamic, in that it can be changed, but also dynamical, in that it changes by itself[10]. This section looks to expand upon the previous by first introducing the idea of *computation-by-navigation* and then showing how passive and active dependency can be added to DOSTE.

4.3.1 Computation by Navigation

Observables in the graph can be accessed by navigation from some starting node (the observational context) and following any number of edges until a resulting node is found. In the object-oriented tradition, nodes, when interpreted as objects, can contain edges that represent operations upon that object. The node an operation edge points to represents a *curried* function and all edges coming from that node correspond to the second parameter being given. Following one of these edges leads to either a resulting value or another function if the function has more than two parameters.

A good example is that of integers where a node represents a number and has edges for addition, subtraction and so on. An example is given in the graph in figure 4.7 which shows an addition operation for the integers 0, 1 and 2. Following the addition edge moves to an intermediate node with edges for all other integers that you may add to the first. All kinds of operation can, in principle at least, be represented by this graph navigation approach but this does require some nodes to be very large or infinite

[10] This use of dynamic and dynamical may be controversial. Dynamical is used for systems that change with time whereas dynamic has a broader meaning often associated with action (of agents).

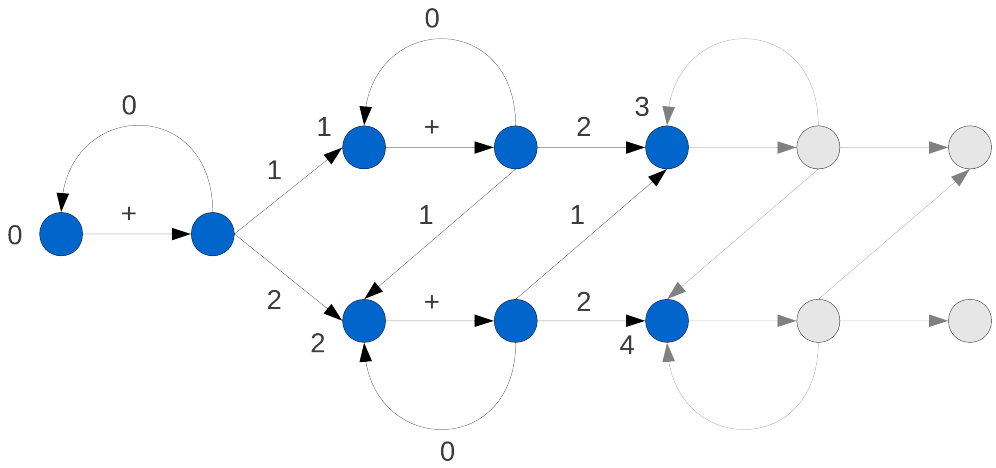

Figure 4.7: A portion of the graph for integer addition.

in extent (i.e. an infinite number of edges) and that there exist a very large or infinite number of nodes. The navigation approach also relies heavily on the fact that node OIDs can identify edges, as will become clear. What this approach to computation results in is an ability to explore logic, arithmetic and functions in the exact same way as exploring the data. There is no difference between data and code, computation becomes a form of observation as well (cf. chapter 6).

The example graph depicted in figure 4.7 can be expressed explicitly using DASM and is shown in the example script in listing 4.14. In practice a full definition of integer addition and other operators is extremely large and is considered infinite, so such a description cannot be given explicitly and must be built-in as a virtual part of the graph[11]. Conceptually, however, it works as shown.

If the definition in listing 4.14 were to be completed for all numbers and operators then arithmetic expressions can be specified as paths to follow through the graph. This takes advantage of the cyclicity of the OD-net graph.

Any arithmetic can now be performed by graph navigation (cf. listings 4.15 and 4.16). The agent that does the following is doing the computation and that agent may

[11] An alternative is to use generic definitions described later as a way of describing potentially infinite graphs in a lazy fashion.

```
0 + = (new);
0 + 0 = 0;
0 + 1 = 1;
0 + 2 = 2;
1 + = (new);
1 + 0 = 1;
1 + 1 = 2;
1 + 2 = 3;
2 + = (new);
2 + 0 = 2;
2 + 1 = 3;
2 + 2 = 4
```

Listing 4.14: DASM definition of addition

```
1 + 2 + 3 + 4 + 5
```

Listing 4.15: Simple arithmetic in DASM

```
5 * (2 + 6) / (7 * 8)
```

Listing 4.16: Arithmetic in DASM

well be the human user, not just an automated process. In a sense it is a little like a structured lookup table that a user or machine can follow to perform some computation, although not restricted to numbers. The graph describes what can be computed by observation, but not what is computed as that is entirely at the discretion of the observer. The example in listing 4.16 shows how sub-queries can be used. Sub-queries work by taking advantage of the fact that edges are labelled with the same OIDs as nodes. So the result of a query is a node that is then used to identify an edge from another node.

Much more will be said about the theory behind computation by navigation in chapter 6. The next example, in listing 4.17, defines the boolean operators. Boolean operators can be described in exactly the same way and because they are small and finite they can easily be given explicitly so are not a built-in virtual part of the graph. In listing 4.17 is the boolean 'and' operation given in DASM:

```
true and = (new);
true and true = true;
true and false = false;
false and = (new);
false and true = false;
false and false = false;
```

Listing 4.17: DASM boolean 'and' operator definition

Listing 4.18 is an example of using the 'and' operator for a boolean expression. It is easy to visualise the computation process by following the graph in figure 4.8 to evaluate the expression in listing 4.18. The node obtained after all edges have been followed is the result of evaluating this expression, in this case '**false**'.

```
true and true and false and true
```

Listing 4.18: DASM boolean logic

Conditional statements can also be expressed using graph queries and will be

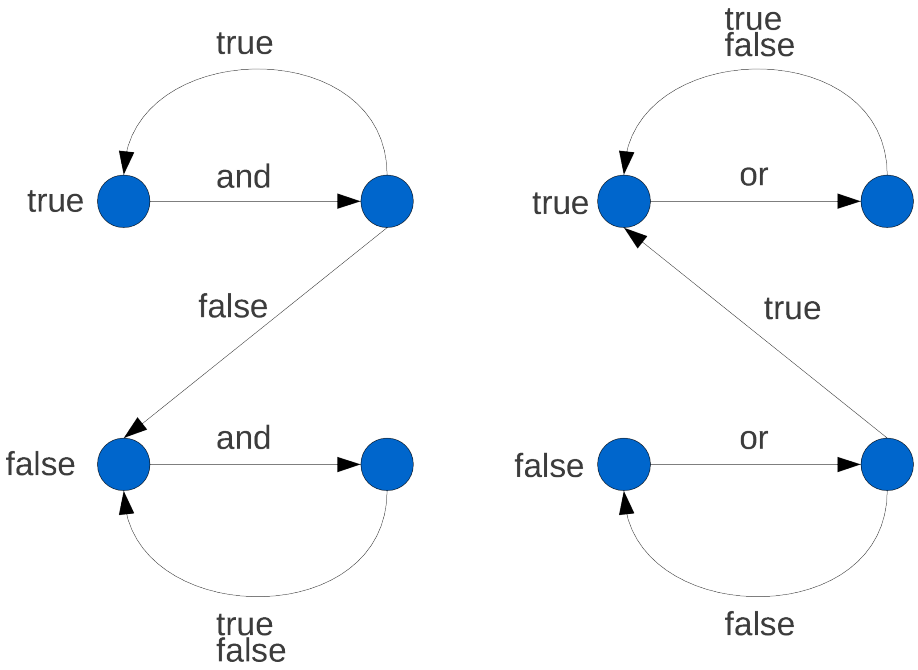

Figure 4.8: Complete graphs showing boolean 'and' and 'or' operators in DOSTE

shown in the next section.

4.3.2 Definitions for Passive Dependency

To support Empirical Modelling's original notion of dependency (cf. §2.2.2), called *passive* dependency here, edges in the DOSTE graph can be given definitions instead of directly pointing to another node. A definition, when evaluated, gives the node that the edge should be pointing to. Since computation is by graph navigation, definitions are described as paths through the graph which are to be followed to "calculate" a resulting node. Whenever any of the edges in the path followed change, either by their own definitions or by agent action, then the edge's definition is marked as out-of-date so when next observed the definition is re-evaluated. This is dependency maintenance in action.

Latent definitions, as opposed to dynamical definitions (cf. §4.3.3), describe passive dependencies and can be given in DASM using a special '**is**' keyword instead

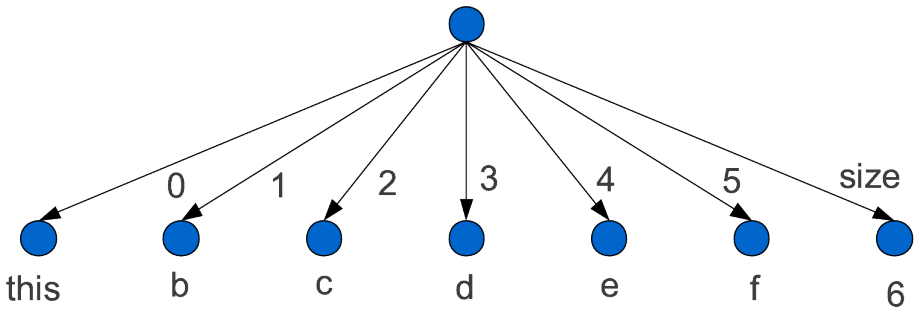

Figure 4.9: DOSTE definition represented as a graph structure.

of '=' as used previously. The simplest form of latent definition is one that acts as a short-cut, as shown in listing 4.19.

```
.a is { .b c d e f }
```

Listing 4.19: Shortcut definition in DASM

So now if any of the edges 'b', 'c', 'd', 'e' or 'f' are changed then the edge 'a' is also updated. Note how the right-hand-side of the '**is**' is surrounded by curly braces instead of plain braces (contrast with listing 4.4), which indicates that the rhs should be encoded as a definition instead of immediately evaluated as a path to follow. Definitions in DOSTE are encoded as nodes and edges, so for example the definition in listing 4.19 becomes the graph depicted in figure 4.9. It is possible in DASM to assign a definition encoding to an observable using '=' with curly braces on the rhs, allowing a definition graph to be explored in the same way as any other structure. Similarly a definition structure could be built manually and then used with '**is**' by giving a path (or node) on the rhs instead of a curly brace definition[12].

A slightly more complex example of a definition would involve performing some calculation. In the example in listing 4.20 a definition is used to always maintain an observable as the square of another.

[12]The curly brace syntax for constructing definitions is syntactic sugar.

```
.a = 5;
.b is { .a * (.a) }
```

Listing 4.20: Definition to square a number

In listing 4.20 'b' would have the value of 25 and if 'a' were then changed to 6 it would automatically and indivisibly become 36. These kinds of definition are clearly similar to spreadsheet formula. Unlike spreadsheets they return nodes that describe structures and so is closer to, but less restricted than, Forms/3 which supports richer types (cf. §2.1.3). With a passive dependency it is not possible to observe a state where the definitions have not been updated (indivisibility of update). Internally DOSTE operates in a lazy fashion, so definitions are only actually re-evaluated upon use when out-of-date rather than as soon as they become out-of-date. This lazy approach saves a great deal of processing and is the same technique that is used in EDEN. In example 4.20 the part after the multiplication has been put inside brackets which states that this part must be done as a sub-query. Without the brackets the definition would read from left to right as a list of edges so would attempt to multiply the dot and apply 'a' to the result. Operator precedence is unlike that in most languages, it is always left to right. This is because of the graph navigation interpretation.

There are two additional things to be aware of with regards to latent definitions and the use of passive dependency:

1. They cannot be cyclic, so must not refer to their own observable on the rhs either directly or indirectly. The example in listing 4.21 shows indirect cyclicity and will fail with an error message in Cadence[13].

2. Performing an assignment does not override the definition and so assignments are completely ignored for any edges with latent definitions.

[13] Cyclicity errors will not cause Cadence to crash, it gives a message and returns *null* as the result of the definition.

```
.a is { .c }
.b is { .a }
.c is { .b }
```

Listing 4.21: Indirect cyclicity example

Definitions may also need to use conditionals to decide what paths to follow. There are no special operators for conditional statements in DOSTE definitions as they are not required. It is possible to construct graph structures using latent definitions and node OIDs to select edges which produce the same effect as a conditional statement. A conditional is used to select an edge in a node and the value that edge points to is the result of the conditional. An example is given in listing 4.22 of such a structure.

```
.ifdemo = (new
    . = (.)
    true is { ..b }
    false is { ..c }
);
.d is { .ifdemo (.a == 1) }
```

Listing 4.22: If-object construct in DASM

In the example the edge `d` is defined to be the value of edge `b` when `a` is 1, otherwise it is the value of edge `c`. Having to construct such structures for every conditional statement would take time and so a syntactic sugar has been added to DASM that enables a more traditional way or writing such *if* statements (cf. listing 4.23). It should be noted that internally all this does is automatically generate the structure shown in listing 4.22.

Notice how, in listing 4.23, inside the true and false parts double dot is used instead of just a single dot. This is a direct consequence of the actual translation of this

```
.d is {
    if (.a == 1) {
        ..b
    } else {
        ..c
    }
}
```

Listing 4.23: Syntactic sugar for conditionals

statement because a single dot refers to the *if* object itself which then has a parent[14] set to be the original context, hence needing double dot to access the original context inside the *if*. Nested ifs are possible, as is *else-if*, and in all of these double dot is required. It always remains only double dot rather than triple because all nested *ifs* still have their parent set to the original context rather than their parent *if*. If-constructs are an example of a Cadence design pattern; there are many more such patterns that have been identified.

4.3.3 Definitions for Active Dependency

Dynamical definitions implement the active form of dependency for describing processes[15]. Instead of referring to the present they describe what the future value of an edge will be. Dynamical definitions can refer to the current value of the observable which they are defining and so allows future values to be defined in terms of the current value. Such definitions can be used as counters and for animation as well as other dynamical systems applications. In DASM a dynamical definition is given using '**:=**' instead of '**is**' but is otherwise syntactically identical. The example in listing 4.24 will

[14] Objects may have an edge that points to its parent object, and usually this is an edge labelled with a *dot*. It may not always be possible to have a single parent, but when cloning is used to generate the object the parent edge is automatically set.

[15] Dynamical definitions were at one time the only type of definition in Cadence until *hiaton* issues where identified that required the latent form of definitions. Both are therefore needed. Hiaton issues are synchronisation problems due to different lengths of dependency chains.

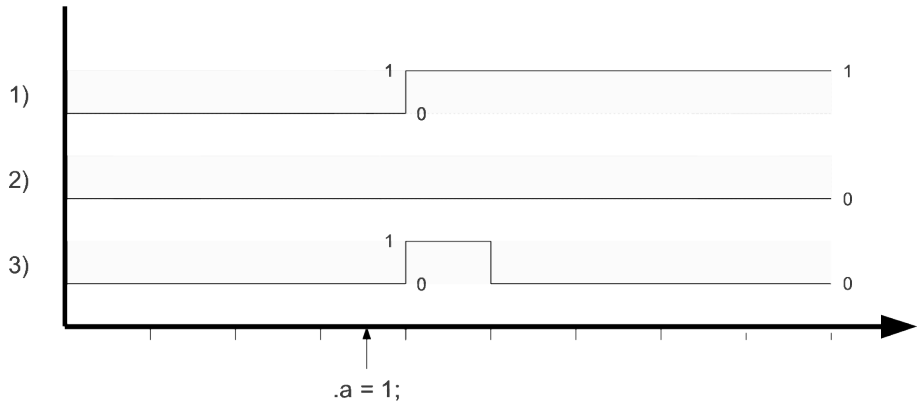

Figure 4.10: Definition comparison over time. Relates to listing 4.25.

count as fast as the machine can by having an edge add one to itself continually. Notice how 'a' must also be initialised to some initial value using assignment. With dynamical definitions it is possible to have a definition and to use assignment at the same time, unlike latent definitions.

```
.a = 0;
.a := { .a + 1 };
```

Listing 4.24: Counting with dynamical definitions

The semantics of the different kinds of definition can be illustrated using the set of possible definitions for 'a' given in listing 4.25 and the graph in figure 4.10 showing how the observable 'a' is affected over time by an assignment for each of the three possible definitions it may have.

```
1 .a = 0
2 .a is {0}
3 .a := {0}
```

Listing 4.25: Semantics of definition types

The first line of listing 4.25 shows assignment which defines 'a' to be 0 now but without saying anything about how it might change. The second line gives a constant latent definition which says that 'a' is 0 now and will always be maintained as 0 regardless of any assignments performed on it. The final line states that 'a' will become 0 in the very next instant but that right now it could be something else. If an assignment is performed when the definition on line 3 is in place then it will take effect but in the very next instant the value will revert back to 0, as is shown in figure 4.10. By default all edges can be thought of as having the dynamical definition in listing 4.26 which gets overridden by the user.

```
.a := {.a}
```

Listing 4.26: Default definition

What the definition in listing 4.26 says is that 'a' will always become whatever it already is and because a dynamical definition is used it is possible to use '=' to change its value. Removing a definition from an observable is not possible, instead the above definition would need to be reinstated to provide the default functionality. Internally definitions are only evaluated if they need to be due to dependency maintenance and as a consequence the definition given in listing 4.26 effectively becomes latent in that no change is occurring until agent intervention.

4.3.4 Generic Definitions

To a limited extent it is possible to describe generic definitions in DASM using the '$' token which gets substituted with the edge label. An experimental feature, it has already shown potential in allowing functional style node descriptions since it can act as a function parameter to be specified as needed. Previously there was an example showing a definition being used to define the square of some number (cf. listing 4.20). An alternative way of describing this operation would be to use a generic definition.

```
.square = (new
    $ is { $ * ($) }
);
```

Listing 4.27: Generic definition to square a number

```
.square 5
```

Listing 4.28: Using the generic square

Example 4.27 describes a generic definition which multiplies an edge node with itself to produce its square. This does not require the edge to be a number so can be applied to any node that has a multiplication edge. In the second example, shown in listing 4.28, the number 5 edge is applied to square and so the result of that expression will be the node 25. The definition is not instantiated until required for a specific edge, at which point the new edge is automatically constructed and the definition evaluated and cached. In other words generic definitions are lazy. '$' will match any edge that does not already exist in the object so if an edge is explicitly given it will override the generic definition. With this mechanism it is possible to provide a base case to an otherwise recursive generic definition. A recursive example is given in listing 4.29 that calculates factorial.

```
.factorial = (new
    0 = 1
    1 = 1
    $ is { $ * (. ($ - 1)) }
);
```

Listing 4.29: Factorial using generic definitions

Generic definitions and their potential are explored further in chapter 6 and is

also discussed as future work. It would be one way of supporting abstraction but is difficult to directly experience through exploration of the graph as it has no concrete instantiation.

4.4 Implementing Cadence

Cadence has been written in C++ and is supported on Windows, Linux and MacOS, now available as an open-source project on github [Pope, 2011]. It may also be, and has been, compiled as an independent operating system[16]. To fully appreciate how Cadence operates an understanding of its evaluation mechanisms and internal representations is useful, especially for dynamical definitions. This section will briefly discuss the architecture of Cadence and will introduce the C++ API used for developing extensions. A few specific extensions are then given, including a 2D development interface, a 3D graphical interface for games and support for network distribution as well as integration with EDEN.

4.4.1 Architecture Overview

The architecture of Cadence is reasonably simple and is depicted in the diagram in figure 4.1. Of real interest, however, is the architecture of the DOSTE core of Cadence and it is this which is to be discussed here. The DOSTE architecture is shown in figure 4.11. Primarily the architecture is about the routing and processing of events in precise ways to achieve the desired result of indivisibility of latent definitions and the correct "behaviour" for dynamical definitions whilst still allowing for agent interaction and observation. All interaction, observation and internal communication is done using events. Agents generate events to modify and observe the system, these are then sent to one of four queues to be routed, at the correct time, to an event handler. Handlers may also then generate events to maintain dependencies. It is the handlers that store

[16] Not available on GitHub.

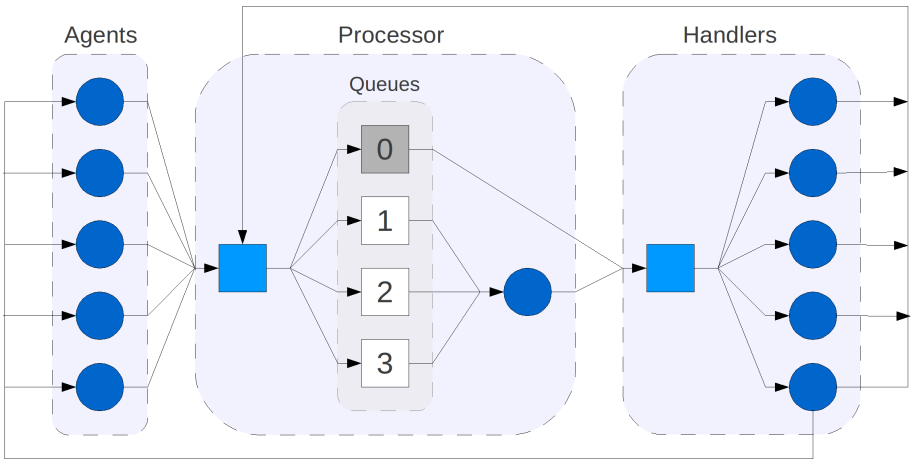

Figure 4.11: DOSTE architecture diagram showing the core components.

some representation of the graph and edge dependencies, with some handlers being used for virtual parts of the graph.

The choice of this particular architecture has come out of considerable experimentation and many different attempts. The nature of the events, the fact that there are four queues for different event types, what types of event there are and how handlers and agents operate is all very precisely controlled and orchestrated to give a reliable representation of an OD-net that can be used for Empirical Modelling. This removes the huge burden of orchestration from the modeller, a problem that Edwards was looking to overcome with Coherence (cf. §2.1.3). There were two additional motivations behind choosing this particular architecture:

1. Network distribution by allowing events to be transparently routed to handlers on different machines.

2. Concurrency by allowing multiple processors to work on processing queued events without causing conflicts[17].

[17] Although not currently maintained, at one time Cadence did support the use of multi-core processors for concurrent processing of events and this worked reasonably efficiently and was reliable. Problems arose when trying to link this with traditional technologies such as OpenGL which did not work well with such fine-grained concurrency.

What is not shown in figure 4.11 is how agents and handlers can be added by loading dynamically linked libraries at any time. Due to this, the core DOSTE part of Cadence is relatively small and consists primarily of the processor part and a framework for adding agents and handlers.

4.4.2 Events and Queues

Event-programming is common in operating systems because it is the most effective way of dealing with asynchronous concurrent communication. It is especially useful for user interfaces where actions can occur at any time in any part of the program but also for inter-process communication (IPC). To think of each definition and agent in DOSTE as a miniature process shows how fine-grained and extensive the potential concurrency is and how important effective IPC is. It is for this reason that events have been chosen since actually modelling DOSTE as a collection of active processes is exceptionally inefficient[18]. Instead DOSTE must be entirely event-driven.

Events are represented in DOSTE as small packets of information that have a standard structure. These structures are passed through the system to be processed at the correct time in the correct place either synchronously or asynchronously. Since the event mechanism is mostly asynchronous it is necessary to have at least one event queue which stores the events before they get processed. In practice however, more than one queue is required because different types of event need to be processed at different times. There are two reasons for this:

1. by grouping all read-only events and write-only events together you can remove the need for some of the concurrency locks and slightly improve performance.

2. more importantly though there is a need to synchronise and get ordering correct in order for the correct behaviour to be observed when dealing with dynamical definitions.

[18]The major operating systems (Windows and Linux) do rely on processes polling for events, but this is acceptable given the comparatively small number of processes involved.

Type	Queue	Description
GET	1	Return the node an edge from a node points to.
GETKEYS	1	Return a list of all edges from a node.
SET	2	Change the node an edge from a node points to.
DEFINE	2	Change the definition of an edge from a node.
NOTIFY	3	Notify an edge that its definition is out-of-date.
ADDDEP	4	Record a dependency on a particular edge.

Table 4.1: Main DOSTE event types

If the system fails to properly order the events it can end up with incorrect and almost random results or no result because some definition fails to get triggered. An example might be if an *add-dependency* event was processed after another *set* event, in which case some definition might not get *notified* of the change that occurred and incorrect values are the result.

Events have the following structure:

type The type of event determines how it is processed

destination The node to which this event is being sent

parameters(n) A number of parameters (max 4), each of which is an OID

result An optional result OID for some events

All events are some action to be performed to a particular node and therefore events have a destination node. It is this destination that determines where the event is sent for processing by a handler. Different nodes get managed by different handlers and this is determined by the OID.

The type attribute determines what action to perform on the destination node. Table 4.1 shows some of the more important types of event. When they are sent they are added to one of three queues depending on the type of event: *write* (SET and

DEFINE), *notify* or *dependency*. Each CPU will go through one queue at a time and process each event. Read events (GET) are always synchronous (don't get added to a queue) but should only happen during the notify queue cycle. Write events get added to the first queue and when processed they may cause notification events to be generated if there are other observables with dependencies on the one being changed. Notify events are added to the next queue which is processed after all the write events have been processed. A notify event may cause a definition to be evaluated which can generate read events and add-dependency events. The read events are performed immediately so that the result of the definition can be calculated. Finally, a notify event will generate a single write event to actually perform the change, but this gets added to the first queue and so will not be processed immediately. This is important because it means all other definitions that still need to be processed can use the old values, otherwise the results would be non-deterministic. Add-dependency events get added to the next queue after notify events so that they are all performed before any write events. If this was not the case then some writes would occur before the dependencies are added and so some definitions will not be correctly notified of a change that does affect them. Once a cycle through the queues is complete the whole process starts again with the first write queue. Each cycle is called an instant[19].

The above description is for dynamical definitions, there is a slight difference for latent definitions because a notification of being out-of-date needs to happen immediately when a change occurs. Latent definitions will evaluate only on-use-when-out-of-date as opposed to dynamical definitions which will evaluate each instant when out-of-date.

A simple example of a definition evaluation can illustrate the flow of events. The example in listing 4.30 describes a definition that centres a button's x-coordinate based upon its width and the width of the window it is in.

As soon as the button x definition is entered by an agent, the agent generates a

[19] An instant is one discrete time step of the dynamical system.

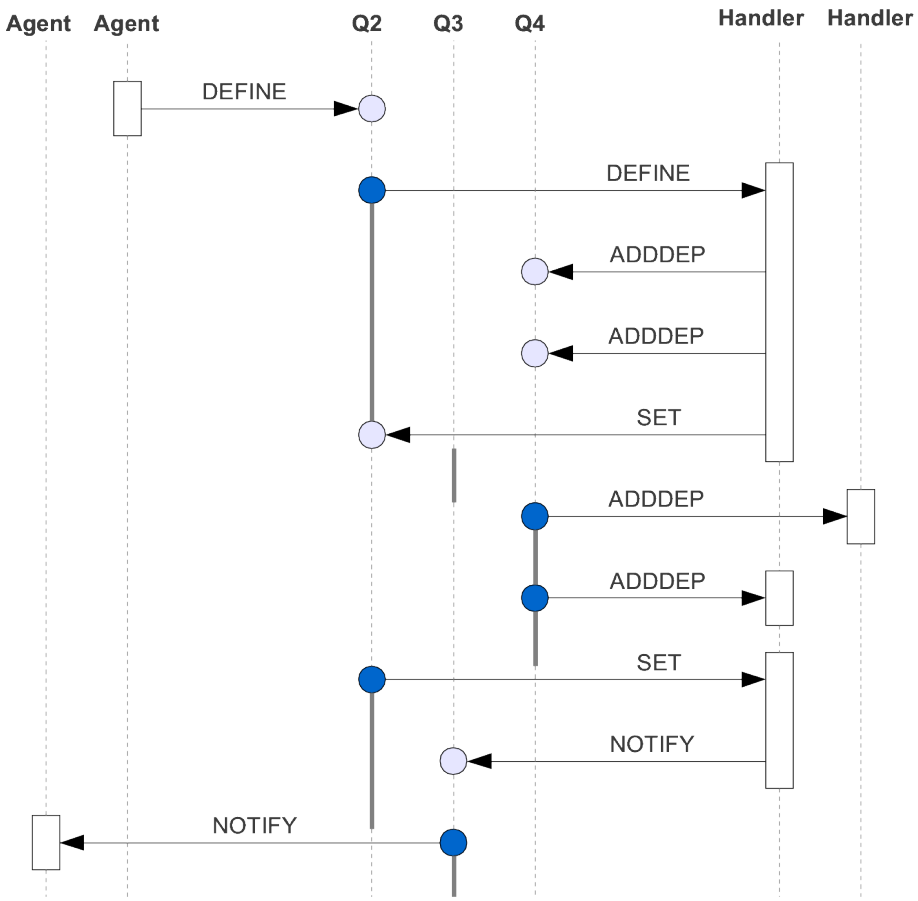

Figure 4.12: DOSTE Event Flow Example

```
.button x := { @window width / 2 - (.width / 2) };
```

Listing 4.30: Button centering example

DEFINE event[20]. Figure 4.12 shows the sequence of events generated by this example, although read events have been left out for clarity. The left most handler in the diagram corresponds to the button node. The other handler is for the window[21]. As can be seen, each queue is processed in order and is highlighted to show when it is being processed. The final consequence is that a NOTIFY event gets sent to another agent that represents a graphical interface being told to redraw the button.

4.4.3 Handlers and Agents

A handler receives events for a particular set of nodes for processing. A handler must internally have some representation of the nodes and edges it is responsible for, as well as keeping track of definitions and dependencies. Each handler gets registered with the event router when it is installed so that the router knows to send events for particular nodes to it.

Handlers may be virtual in that they do not actually store information but generate it on-the-fly or send it over a network for processing by a remote handler. Some handlers may connect directly to hardware. The handlers available in Cadence by default are:

Local Stores nodes and edges in memory.

Network Forwards events to another machine.

Numbers Simulates a virtual graph for numbers.

IO Maps edges to IO ports on the machine.

Files Maps file system into the graph.

Agents Manages agents and forwards events to them.

Custom handlers can be added at any time to extend the system. The last handler listed above is of interest, it represents all agency. Although the diagram in

[20] Although the agent first constructs an object to represent the definition that was entered.
[21] In practice both of these handlers are likely to be the same.

figure 4.11 shows agents and handlers as being separate, this is not actually the case. Agents are managed by and contained within an agent handler. This has been done so that agents can take advantage of the existing event mechanisms to receive notifications of changes that trigger them.

4.4.4 C++ API

The C++ API constructed for Cadence will be briefly discussed since this is a key part of its flexibility. The main focus for the API was to make the addition of new agents as easy as possible so that new functionality could be added by anyone with basic C++ knowledge. To fit with the always running interactive nature of Cadence it is also possible to add these new agents at any time whilst the environment is running.

C++ comes with a large number of features for customising the language. This includes operator overriding, templates and macros that are used extensively in the API. An agent in Cadence is an object in C++ which can respond to changes that occur in the graph. It does this by having event handlers which get called when specific observables change. An agent is then able to observe and change the graph using a simple mechanism as well as use any other C++ libraries to, for example, draw graphics on the screen.

Each agent in C++ is an instance of a class that describes that type of agent (cf. listing 4.31). DOSTE provides a run-time type system which enables all these agents to be automatically constructed as required. To achieve this each agent class (or agent type) is registered with DOSTE and given a label so that Cadence can look at a graph node and determine what type it should be[22]. When another agent requests an agent object from a particular node it will look at the type attribute and find the appropriate class to use to make an instance of that object. In this way the programmer does not need to know the class to use as the system will determine this at run-time. In addition, if an instance for that object already exists then that is returned instead of

[22] Ideally this would be done as duck-typing but is currently done by checking an explicitly given label identifying its type.

```
class Assigner : public Agent {
    public:
    Assigner(const OID &o): Agent(o) { registerEvents(); }
    ~Assigner() {};
    OBJECT(Agent, Assigner);

    BEGIN_EVENTS(Agent);
    EVENT(evt_condition, (*this)("condition"));
    END_EVENTS;
};

OnEvent(Assigner, evt_condition) {
}
IMPLEMENT_EVENTS(Assigner, Agent);
```

Listing 4.31: C++ agent example

a new object being created. With this the programmer does not even need to worry about constructing or deleting agent objects. A benefit of this approach of automatic run-time typing and construction is that dependencies between modules can be removed since one module does not need to be aware of another in that it does not need to know about specific classes in other modules. It is an example of dependency-injection where the OD-net describes what C++ objects to construct.

4.4.5 Graphical Interfaces

Using the provided C++ API, two different graphical interfaces have been developed for Cadence as modules by providing a collection of agents. Neither of these are a focus of this work and are there to enable visualisation and interaction but have not been developed to be especially user friendly. EUD and HCI research would be needed to find a more direct and friendly way of manipulating the Cadence OD-net[23].

[23] Self, Forms/3, Subtext and many others provide inspiration for how this may be done.

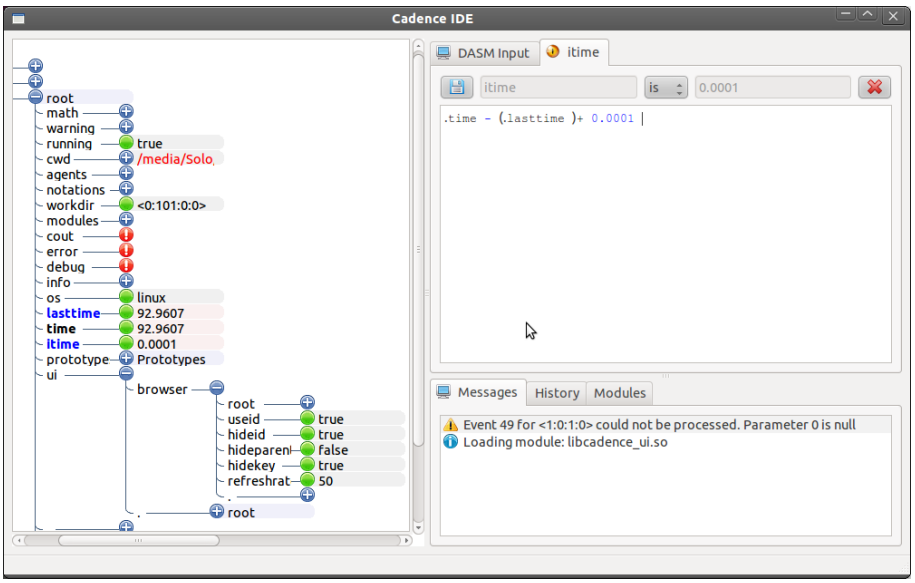

Figure 4.13: Cadence User Interface Module

The first is a development interface, as shown in figure 4.13, which shows a representation of the OD-net as a tree structure and allows for the input of DASM statements to manipulate the graph. The visual representation of the graph can also be manipulated in basic ways to make changes. The OD-net visualisation is live and will be immediately updated in the presence of change, either agent caused or dynamical definition caused. It is this interface which is used by model developers to experiment with the artefact.

The second is a 3D visualisation module developed for the Warwick Game Design society at the University of Warwick. This is a student group who make computer games and at one time it was decided that Cadence, with a suitable set of extensions, could be used to develop games. These extensions involved support for OpenGL windows, animated models, sound effects, sprites, lighting, height-maps and collision detection, as well as for input devices that include the Wii-remote. Much more can be seen of this game library in the next chapter.

4.4.6 Other Extensions

There are other extension modules that have been developed for Cadence. These illustrate the diversity of possible extensions to Cadence and enable Cadence to connect with existing technologies or be used by other software projects in ways that EDEN could not. Below is a list of extensions developed during this work:

EDEN A version of EDEN has been adapted to run as a module of Cadence. There is a communication mechanism between them. More is said on this in §7.2.

XNet Provides a handler for connecting multiple instances of Cadence on multiple machines together in a transparent way by routing events between the machines.

Agents Programmable agency by describing individual actions when certain conditions become true. There is some similarity between these micro-agents and the *actions* found in the ADM [Slade, 1990, p.28].

Web A module that allows web servers to connect to Cadence to provide a web interface. This is currently work-in-progress.

Chapter 5

Cadence Models and Examples

The Cadence prototype described in the previous chapter is best illustrated and evaluated by developing a variety of models within it[1]. In this chapter, the qualities of Cadence will be illustrated with reference to a few of the many models that have been developed both by the author and in collaboration with others. Each model is used to demonstrate some characteristic of Cadence that is relevant to future discussions and to show how it relates back to the objectives of the thesis, the principles of EUD and EM and the issues identified in §3.3. The scripts of the models discussed here are given in Appendix A. It should be remembered that the main comparison is with existing EM tools although some aspects demonstrated by these examples are novel in their own right and may benefit not only plastic applications but traditional applications as well.

5.1 Stargate

There are an increasing number of approaches to composing programs in a flexible way, usually involving XML to link certain components at run-time rather than embedding these structural dependencies into the source code (dependency injection). Cadence can be used for this purpose and the best way to show this is to look at how the Warwick

[1] These models were developed as the Cadence tool was itself being developed and so helped to keep Cadence grounded in practical and plastic applications.

Game Design library, introduced in §4.4.5, is used in Cadence. The game library consists of many different C++ classes to provide specific functionality such as loading textures into memory ready for drawing on the screen using OpenGL. These classes have been kept as independent as possible from each other in code. As a consequence they can be combined in interesting ways at run-time as part of a model. Each component is in fact an agent and observes specific nodes and edges in the Cadence environment, with dependencies being used to link them into models. To demonstrate this a model of the Stargate[2] is used which makes extensive use of the game library and glues around 200 components together to produce all the visual effects. The Stargate model is also an excellent example of the benefits of having a semi-structured OD-net as opposed to the flat OD-net found in EDEN, and of using dynamical definitions.

A Stargate is a fictional artefact from several films and a television series that generates a "wormhole" between planets for fast transportation [Wikia, 2011]. Originally the model was developed to test out the graphics capabilities of the game library whilst also working out the visual and animation details for a game a student member of the Warwick Game Design society had wanted to make involving a Stargate[3]. Figures 5.1 and 5.2 show the resulting Stargate model and the Cadence development interface being used with it.

To generate a model as visually complex as the Stargate involves many components including textures, materials, geometry, pixel shaders, cameras, light sources and a scene manager. In addition to this are the widgets that allow the whole scene to be drawn within a moveable, resizeable window along with any other models. The structure that results from combining all of this has over 200 agents observing the Stargate alone. A diagram is given in figure 5.3 to show how these components relate. The diagram shows all significant components and the basic structural relationships, but does not include any latent or dynamical dependencies. A node shown in grey has

[2]The Stargate model was initially developed in June 2008 but has been updated since then due to changes in Cadence.

[3]Sam Gynn is the original developer of the actual Stargate model used in this project, although most DASM structures and animation were constructed by the author of this thesis.

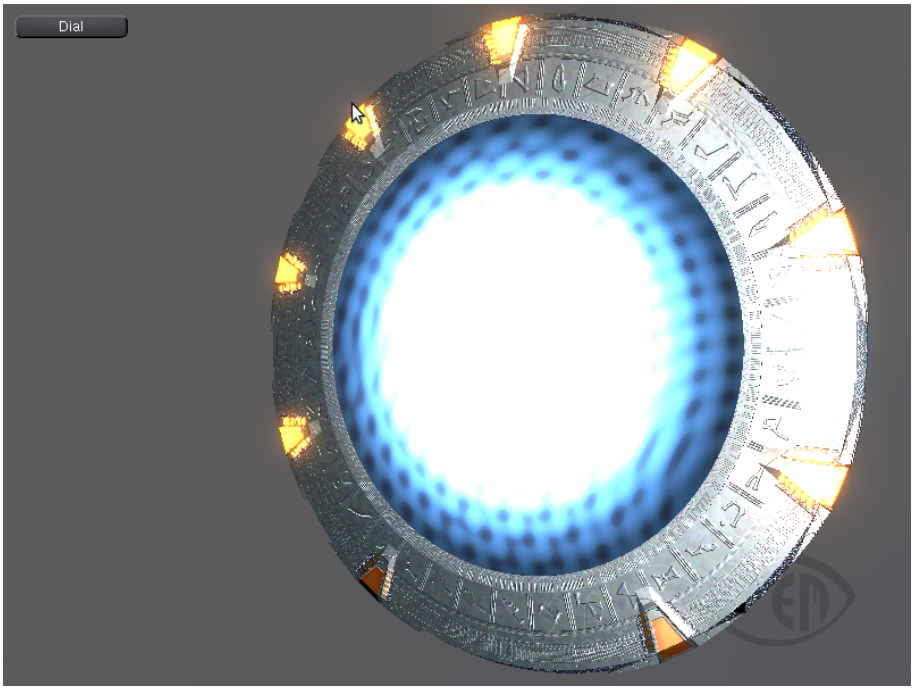

Figure 5.1: The Stargate model

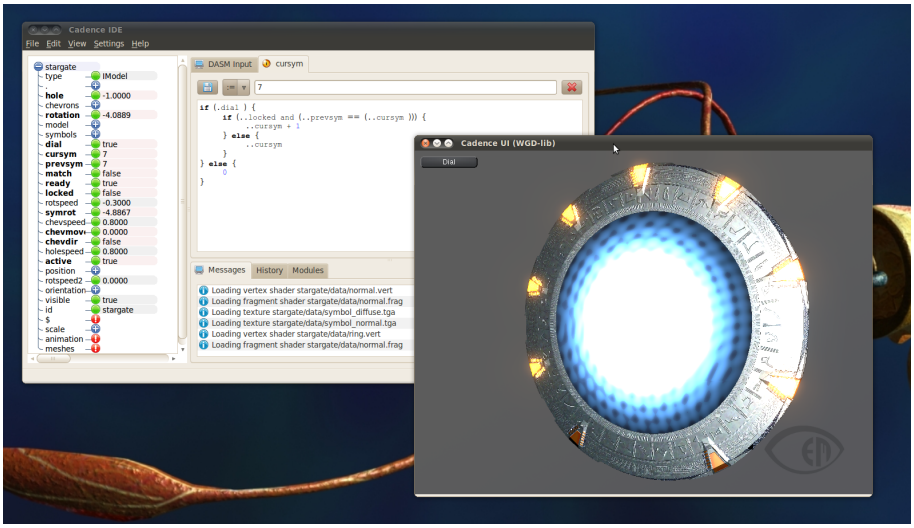

Figure 5.2: Stargate model with the Cadence IDE showing a selection of the Stargate observables

a C++ game library agent associated with and observing it (cf. figure 4.1) to generate OpenGL commands or provide some other functionality based upon the properties of the Cadence node, something explored further in §5.1.1 below. The highlighted region a) is expanded in figure 5.5 to give an idea of the dependencies that also exist between components. Region b) is expanded in figure 5.7. These diagrams could conceivably be automatically generated from information contained within the model, although in this case they have been drawn manually. In existing EM tools there is support for the drawing of dependency diagrams[4].

5.1.1 Shader Composition for Bloom Effect

Focussing for the moment on how shaders[5] in the Stargate model have been combined to produce a HDR (High-Dynamic Range) bloom effect [Kalogirou, 2006] illustrates how Cadence can act as *glue*. Figure 5.5 shows a more detailed view of this part of the model. Bloom is the apparent glow of bright objects in a scene which is illustrated in figure 5.4 where different values and configurations have been used to adjust the strength of the effect. The process of generating the images in figure 5.4 itself illustrates both the liveness (cf. EUD principles in §2.1.1) and experimental characteristics of Cadence models. To achieve the effect it is necessary to render the scene multiple times using a variety of pixel shaders and settings, in this example there are 9 steps involved. Figure 5.5 shows these 9 steps as a chain of nodes that have various dependencies between them, the dependencies being shown by the red dashed edges. The class name of the associated C++ agents is given (e.g. *RenderTarget*). To fully demonstrate how Cadence has been successfully used in this way the remainder of this section details each step of the process.

Step 1: The first RenderTarget object (cf. the bottom node in figure 5.5) will take a scene, specified by the Scene object, and a camera, which gives perspective

[4]This refers to the DMT as mentioned in the footnotes of §3.3.1. In chapter 7 the DMT has been modified to draw graph structures.

[5]Shaders are small programs that are run on the highly parallel GPU of the graphics card to control the rendering process in custom ways.

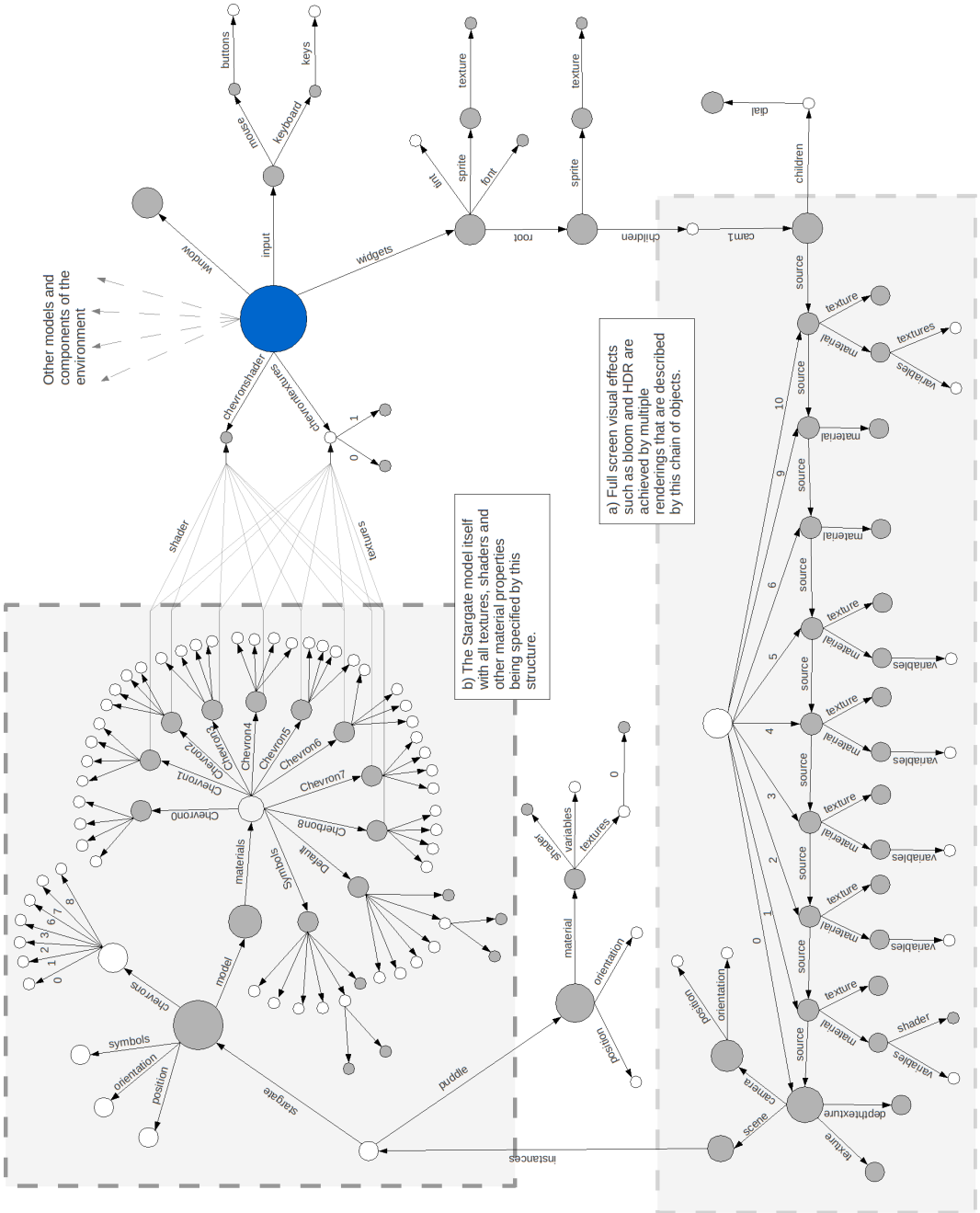

Figure 5.3: Graph structure of the Stargate model. Grey nodes indicate the existence of a C++ game library agent attached to that object. The blue node is the root node. Only structural relationships are shown, no latent or dynamic dependencies.

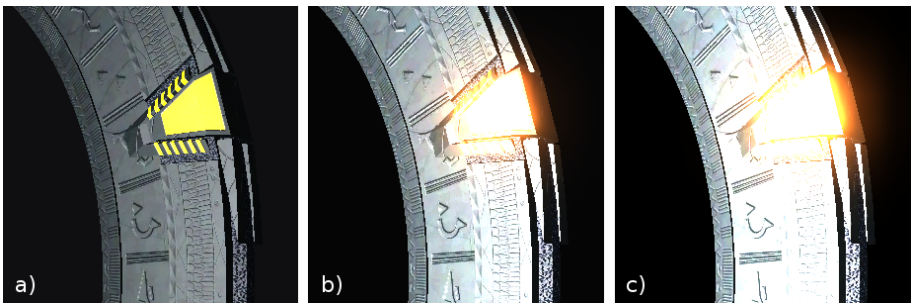

Figure 5.4: Bloom configurations. Each image denotes a different configuration of the shaders that can be made while the model is active

and position information, and render that scene into a Texture object. In this case it also renders depth information into another texture. The listing in 5.1 shows how the destination texture is described in Cadence.

```
texture = (new
    type = Texture
    width is { @sgwidget children cam1 width }
    height is { @sgwidget children cam1 height }
    compress = false
    hdr = true
    nearest = true
    clamp = true
)
```

Listing 5.1: Description of a texture

Dependencies have been used in a basic way here to connect the width and height of the texture object to the width and height of the window in which the resulting image will be displayed. This means that if the user changed the resolution of the 3D window then all buffers would also change by dependency. It is a good example of where dependency can be used to maintain an otherwise troublesome relationship and simple dependencies of this form can be found in other user interface technologies (e.g. WPF

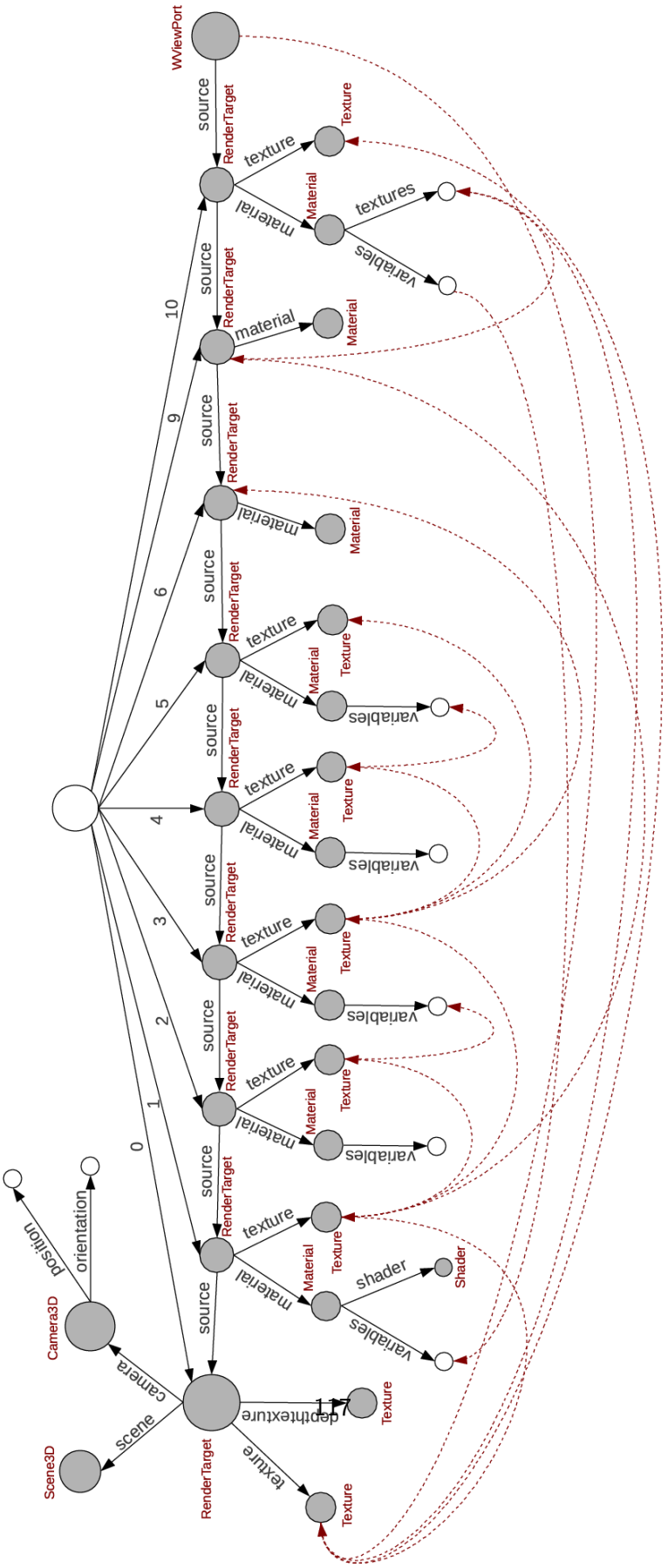

Figure 5.5: Stargate bloom component structural and logical dependencies

dependency properties). Example 5.1 also includes several properties used internally by the C++ agent, such as *hdr* that tells it to use floating point colour values instead of 8-bit integers so that a much higher range of brightness and colour can be represented. Relating back to the issues identified with EDEN (cf. table 3.2), the texture object in 5.1 is an example of a richer type and so shows how the Cadence approach helps resolve problem B3, that of primitive types being inadequate. Since the texture object is also self-describing and groups related observables together, it helps with problem B6 as well.

Another interesting object is the Camera object as this contains definitions to map from keyboard and mouse inputs to camera motion through the scene. Good examples are the *deltax* and *x* definitions shown in listing 5.2. These show potentially complex and dynamical dependencies. Whilst these are not animating dependencies, they are self-referent so would not be possible in EDEN (cf. problem C1 in table 3.2) where an agent would be required to move the camera instead.

```
deltax := {
    @math sin (.orientation y - 1.5707) * (.dleft +
    (.dright)) + (@math cos (.orientation y - 1.5707) *
    (.dup + (.ddown)) * (.scalexz))
}

x := { .x + (@root itime * (@camera deltax)) }
```

Listing 5.2: Camera motion definitions in Stargate

Step 2: After the scene has been rendered the next step is to apply a threshold to it in order to extract only the bright parts of the scene. Another *RenderTarget* is used with a specific pixel shader that either renders the corresponding pixel or renders a black pixel based upon a simple threshold test. The resulting texture object is also scaled down to half the size of the original scene which is achieved by the dependencies for *width* and *height* given in listing 5.3. Figure 5.6 shows the results of each step including this one.

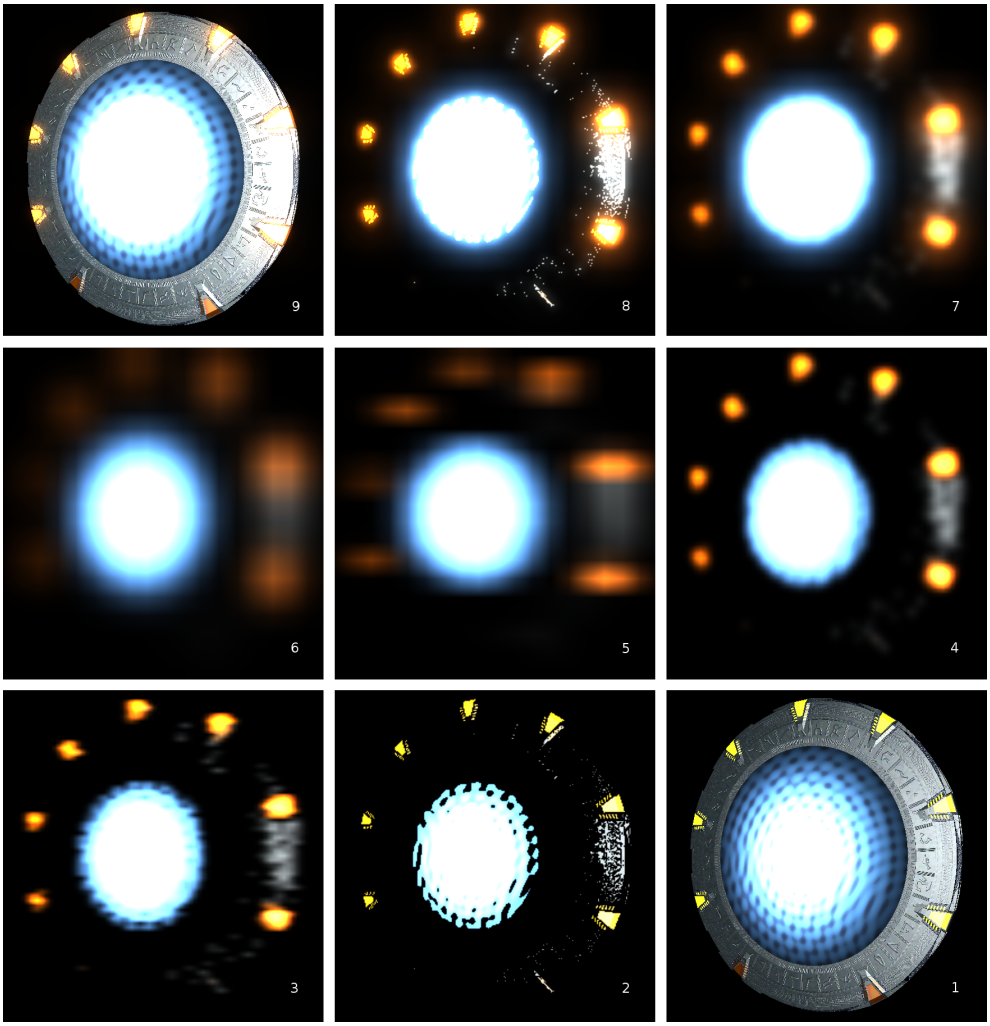

Figure 5.6: The 9 steps of the bloom effect

```
width := { @bloom 0 texture width / 2 }
height := { @bloom 0 texture height / 2 }
```

Listing 5.3: Width and height scaling using dependency

The threshold value is given to the shader as a shader variable (cf. listing 5.4).

```
variables = (new
    brightThreshold := { @bloom 10 material variables
                brightThreshold }
)
```

Listing 5.4: Shader variables connected by dependency

Steps 3,4,5 and 6: All of these steps involve blurring the image and reducing the resolution. Steps 3 and 5 blur in the horizontal with 4 and 6 blurring vertically. The result in step 6 is a blurred image one eighth the size of the original rendered scene. Again dependency is used to define what the width and height of the resulting texture should be, along with specifying a shader variable used for the blurring process to select which dimension and how much to blur (cf. listing 5.5).

```
dy := { 1.0 / (@bloom 2 texture height) }
dx = 0.0
```

Listing 5.5: Definitions to control blurring

Steps 7 and 8: These two steps combine first the blurred images from steps 4 and 6 and then combines the result of that with the original thresholded image of step 2.

Step 9: Finally the last stage combines the blurred thresholded image of step 8 with the original HDR scene rendering of step 1. This involves the use of a special HDR shader to convert the floating point HDR colour information into 8-bit colour values by using various settings such as exposure. This process is called tone mapping. Once complete this is used by a widget agent to draw to the screen.

Note that the images in figure 5.6 showing each of the above steps were generated by trivially altering the structure of the DOSTE graph so as to bypass certain steps

and directly draw the image to the screen. The DASM statement in 5.6 was used to achieve this and clearly illustrates the ease with which such structures can be constructed and manipulated for different visual effects. Such manipulation shows how Cadence ameliorates problems B1 and B5 (cf. EDEN problems, table 3.2) by allowing whole sub-graphs to be switched around rather than individual observables being redefined manually or by loading new scripts. There are actually many more steps included in the model that have been bypassed as they did not seem to add much to the image quality. Looking at figure 5.5, the numbers on some edges jump from 6 to 9 but numbers 7 and 8 do exist, as do numbers 11-15. This is an example of subjective exploratory experimentation, something which was done throughout the development of the Stargate model and already indicates the Empirical Modelling nature of modelling with Cadence.

```
.widget root children cam1 source = (@bloom 0);
```

Listing 5.6: Bypassing bloom steps by changing the graph

5.1.2 Applying Materials to a Model

Another aspect of the model which is of interest here is region b) in figure 5.3. This part of the model describes the material properties of the actual Stargate model. It also contains all definitions relating to animating the Stargate which are detailed in the next section. A more detailed diagram of region b) is given in figure 5.7 which, as with figure 5.5, includes red dashed edges to show where there are dependencies between components[6]. As before, the grey nodes indicate the existence of a C++ object being associated with those nodes and the type of C++ object (the class name) is also given. This section will go through how a model is described in Cadence using the Warwick Game Design Library in order to further show the power of the semi-structured OD-net approach used in Cadence. More technical detail is given here as to how this relates to

[6]The dependencies shown in the diagram are only a small subset of the actual dependencies involved.

the C++ agents involved.

Entities that are to appear in the 3D scene need to be associated with one of several C++ agents. These agents include: IModel, IPrimitive, ISprite3D, ILine and IHeightMap. If they are not associated with such an agent they will not get drawn since no agent is observing them. The C++ agents behind these expect to find information in the OD-net regarding position and orientation of the entity along with certain other information specific to each type of entity which they can then use to correctly draw the entity using OpenGL commands. In the Stargate model both IModel and IPrimitive are being used. The IModel agent is used for the Stargate itself and expects to also contain an edge, labelled as 'model', pointing to a node associated with a Model agent (cf. figure 5.7). IModel acts as a particular instance of a specific 3D model that is specified in the Model agents node[7]. The Model agent is associated with a node that has properties which tell it what model file to load from disk as well as what the material properties of various parts of that model are to be. Materials in a model file are given specific names and these are the same names used as edges in the OD-net inside the materials node. Each material can contain one or more textures, a vertex and pixel shader and other properties such as diffuse and specular colours.

In the Stargate model each chevron in the model is associated with a specific chevron material (cf. figure 5.7). Each chevron material makes use of a special chevron vertex and pixel shader. These have two effects. 1) The vertex shader can offset a chevron by a specified amount given as a shader variable called 'position'. 2) The pixel shader can alter the intensity of the orange colour of a chevron to, in effect, turn the lights on and off, again controlled by a shader variable which is called 'on'. These shader variables are in the OD-net to be observed by a shader agent and can, as a result, be given definitions that control how to move the chevron and when it is switched on and off. An example of the shader variables for one of the chevrons is shown in listing 5.7.

The example shows that both the 'on' and 'position' variables are given simple

[7] Internally the two *agents* do communicate with each other but which agents are connected to which comes from the OD-net.

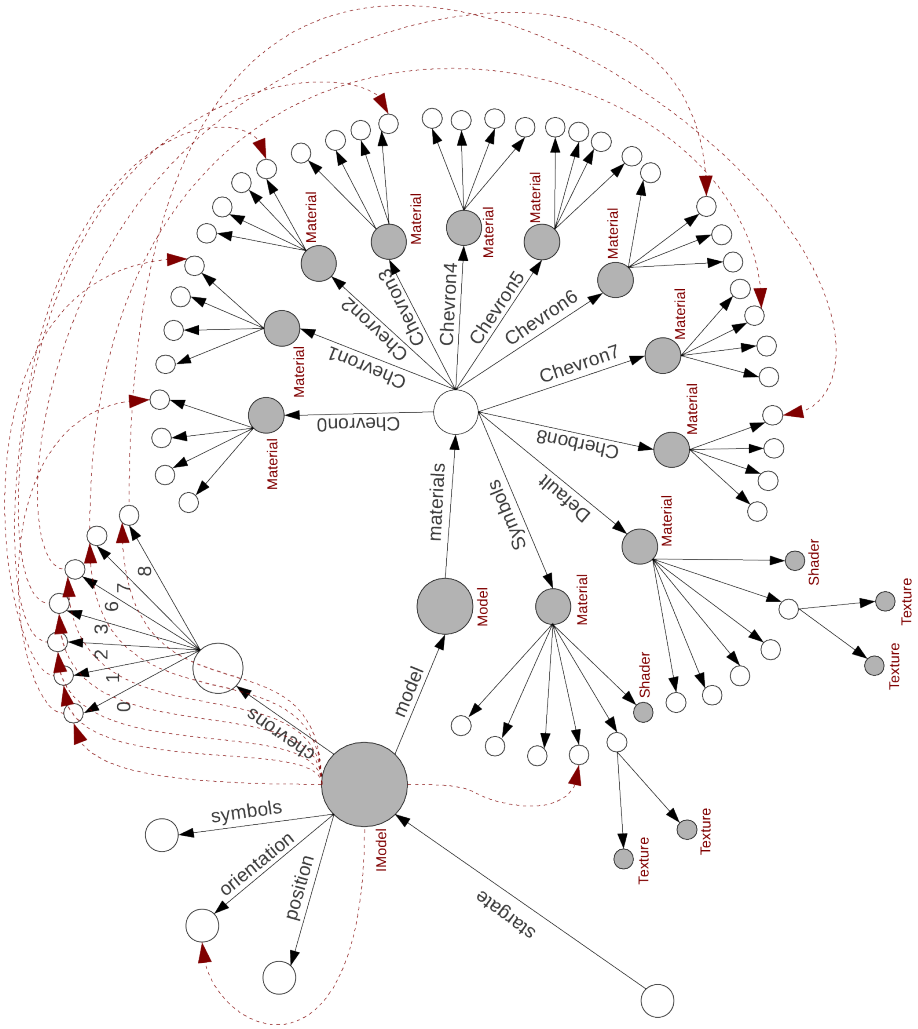

Figure 5.7: Stargate model component structure with selected dependencies shown.

```
variables = (new
    colourMap = 0
    normalMap = 1
    on := { @stargate chevrons 0 on }
    position := { @stargate chevrons 0 position }
)
```

Listing 5.7: Stargate chevron shader variables

Figure 5.8: Left: Chevron position set to 1. Middle: Chevron position set to 0. Right: Chevron on set to 1

short-cut definitions that link their values to some other part of the graph. This has been done so that these properties can be controlled from elsewhere, removing the need to know exactly where to find them and hiding all the other material details. Doing this allows a degree of abstraction and shifting of observational context (cf. B5 in table 3.2) by moving away from concerns about materials and towards higher level controls of the artefact (later used for animation) which demonstrates potential for scalability (cf. B2 in table 3.2).

The scripts in listings 5.8, 5.9 and 5.10 correspond to the images shown in figure 5.8 which illustrates what happens when these variables are changed. Small experiments such as these are performed by the modeller during development to either test functionality (post-theory experiment), or to compare with the referent Stargate or, importantly, to rehearse an animation sequence to be developed later (pre-theory experiment).

```
@stargate model materials Chevron0 variables position = 1.0;
```

Listing 5.8: Fully extending a Stargate chevron

A significant aspect of all this is the simplicity with which OpenGL shaders, materials and so on have been linked with Cadence so that properties and variables can

```
@stargate model materials Chevron0 variables position = 0.0;
```

Listing 5.9: Retracting a Stargate chevron

```
@stargate model materials Chevron0 variables on = 1.0;
```

Listing 5.10: Lighting up a Stargate chevron

be manipulated in a direct and exploratory manner. It enables a far richer model of state to be developed than is currently possible in EM tools, largely due to the semi-structured nature of the OD-net in Cadence, without which such rich agent observations and manipulations would be rather more difficult. Not only is it an improvement on EM tools but is also an improvement on existing game libraries and graphical tools since it enables typically hidden properties to be observed, manipulated and, most significantly, to be connected dynamically with other parts of a model using dependency[8]. Although it must be remembered that this is a generic system not designed specifically for games and 3D graphics as these are only modular add-ons. See Appendix B for C++ code examples of how objects such as shaders are implemented.

5.1.3 Use of Dynamical Dependencies

As well as the glue aspect of Cadence, the Stargate model illustrates the use of dynamical definitions for animation purposes, something not possible in any existing EM tools. These dynamical definitions have been used to rotate the dialling wheel and move the chevrons etc. This, combined with additional latent definitions describing the overall state of the Stargate, enables an animation sequence to be developed in what is hopefully an intuitive way. Such use of dynamical dependency goes beyond simple animation and in fact becomes a fundamental part of the model itself since the logic of the dialling

[8]Sophisticated modelling tools and game engines do provide access to a large number of properties interactively but not to the extent and in the way that Cadence can.

sequence depends upon the rotation. In this section, the process of developing the Stargate dialling animation will be given as an example of how such models can and should be developed using the Cadence tool. The description is given as a sequential story which gradually forms the resulting dialling animation through a process of exploratory experimentation where the modeller is learning about the artefact and referent as they construct it[9].

Before the modeller attempts to develop the animation, observables for visualising the animation, which includes some of the shader variables identified in the previous section, should be available. However, it is not always possible to identify all such observables in advance. These observables will be used to alter the appearance of the model to visualise the animation, something which the modeller can experiment with manually first before attempting to find the correct definition to produce the effect. The process starts with a simple animation of the dialling ring, which is similar to an old fashioned telephone. The ring is made to rotate when the dialling animation is active, as shown in listing 5.11.

```
rotation = 0.0
rotation := {
    if (.dial) {
        ..rotation + (..rotspeed * (@root itime))
    } else {
        ..rotation
    }
}
```

Listing 5.11: Dynamical definition to rotate the dialling wheel

The definition in 5.11 initialises the rotation to 0.0 and then says that it will become itself plus some small amount so that it will continue to move as long as 'dial'

[9]Constructionist learning is a topic of interest for Empirical Modelling which sees its tools and concepts as supporting constructionism [Beynon and Harfield, 2010].

remains true[10]. In the referent (i.e. the Stargate that features in the television series and films) the rotation stops when the correct symbol on the dial wheel lines up with the current chevron on the outside edge. This is repeated for each chevron until all of them are *locked* to a particular symbol and the gate becomes active. So the next step might be to have a set of symbols to dial and some concept of what the current chevron and symbol should be. This information could then be used to detect when the rotation is correct for matching a symbol with a chevron. A possible definition for a 'match' observable is as given in listing 5.12; this was found by a mix of experimentation and off-line (in the head) reasoning.

```
symrot is { 0.1611 * (.symbols (.cursym)) - (0.6981 *
    (.cursym)) }
match is { .rotation < (.symrot + 0.1) and (.rotation >
    (.symrot - 0.1)) and (.dial) }
```

Listing 5.12: Observables to check for symbol alignment

Given this new 'match' observable the modeller could now modify the original rotation definition, shown in listing 5.11, so that the condition for when it rotates uses 'match' and so stops rotating when it matches. The if-condition would then be replaced by the one shown in listing 5.13.

```
if (.dial and (.match not)) {
        ...}
```

Listing 5.13: Modified rotation condition

Again with reference to the Stargate in the television series, when a match occurs the chevrons move and then light up to indicate that they have become locked to that symbol. So now that matches can be detected the next step in developing the animation

[10]The 'itime' observable converts the equation from discrete instants into real-time and is the time in seconds between each *instant*.

sequence is to move the chevrons in the appropriate way and then light them up. Shader variables already exist for this purpose, as shown in the previous section. For the chevron to move, an observable within that chevron needs to go from 0.0 to 1.0 and back to 0.0 again, at which point the light goes from 0 to 1. Three observables can be created that give this "behaviour"[11]; these are shown in listing 5.14.

```
chevdir := {
    if (.match == false) {false} else {
        if (..chevmove > 0.9999) {true} else {
            ..chevdir
        }
    }
}

chevmove := {
    if (.match) {
        if (..locked) {0.0} else {
            if (..chevdir) {
                ..chevmove - (..chevspeed * (@root itime))
            } else {
                ..chevmove + (..chevspeed * (@root itime))
            }
        }
    } else {0.0}
}

locked is { .chevmove < 0.0001 and (.chevdir == true) }
```

Listing 5.14: Animating the Stargate chevrons

The 'chevdir' observable is used to choose which direction the chevron should be moving in. The value of the observable 'chevmove' actually goes from 0.0 to 1.0 or

[11] As far as this can be called a behaviour since it is not associated with agency but rather an instability in the current state.

1.0 to 0.0 depending upon the value of 'chevdir'. The final observable, called 'locked', indicates that the movement animation has completed and that the chevron should be considered locked. These observables now need to be associated with actual chevron shader variables to produce the visual effect. Each chevron has a definition for its 'position' and 'on' observables that checks if it is the current chevron and if it is then it takes on the values of 'chevmove' and 'locked' respectively. The whole notion of "current chevron" is an example of the shifting of contexts, which is difficult to achieve in existing EM tools (cf. issue B5 in table 3.2).

With a few additional modifications this animation can be made to occur for all chevrons and when all 7 are locked the gate can be considered active, at which point all dialling animation stops and the central puddle effect can be started. A complete listing of the Stargate script can be found in Appendix A.

Whilst the modelling process described here seems prescribed and simple, in practice each of the above steps required much trial and error exploratory experimentation before the definitions were seen as faithful to its referent. Such experiment is possible because of the flexible, resilient and observable nature of the model and it shows that modelling in this EM style is possible with Cadence. It also shows how dynamical definitions enable far more to be described in dependency form without resorting to external agent actions. The benefit is that such an animation can be started and stopped by an agent but that the agent does not need to be concerned with actually updating rotations and current symbols. This further reduces the complexity of agents in the same fashion as latent definitions already have in EDEN.

5.1.4 Stargate Summary

Such modularity and flexibility to link into more traditional software components is missing in the existing Empirical Modelling tools and is also an example of how EM thinking may actually be applied more widely to make these sorts of problems easier to manage. The use of dependency with a graph structure as a glue between software

components is novel and as can been seen with this example is a powerful and useful concept. This applies beyond just the OpenGL game library components, although that is all that is demonstrated here. Besides illustrating how it can connect with traditional software libraries and act as a glue between components, this Stargate example shows the power and benefits of dynamical dependencies (issues C1 and C2) as well as the necessity of having structure within the OD-net. Without structure such modularity would be exceptionally difficult. It is worth noting also that many of the problems found in EDEN such as aliasing problems (A2 and A3), searching of observables (B1), shifting of contexts (B5), dealing with complex types (B3 and B6) and so on, were not encountered in the development of this model (cf. §3.3). The model is also an example of an EM style modelling activity (cf. §2.2.4) which will be discussed further in chapter 7. Also note that this model has been used for teaching EM with Cadence, something that will also be discussed further in chapter 7.

5.2 Hardware Device Drivers

A few example models have been developed which explore concepts that potentially relate to operating systems and computer hardware[12]. Considering how EM and its concepts can be applied at a more fundamental system level was an early objective of this doctoral research. An early version of Cadence was developed which could run independently on a machine as its own operating system. The intention here was to develop device drivers and all other parts of an operating system within a Cadence-like environment with prototype-based object-like structures and dependencies being the fundamental concepts. There are two examples worth including here, 1) a keyboard driver and 2) a terminal output driver. Both examples illustrate the potential of such an approach as well as the associated problems.

The keyboard driver was successfully developed within the early operating system version of Cadence. It involves reading and writing to IO ports, responding to interrupts

[12] These drivers were developed in November 2007, before the current Cadence tool was developed.

and then translating the key code into an ASCII character for further processing later on. All of this was achieved via the dependency mechanism since the IO ports were virtually mapped into the OD-net, as were the interrupts. For example, when an interrupt occurred it would set a property to true in DOSTE and this would cause other dependency formula to automatically update by checking the values of the IO port properties. When the key code changes as a result of the interrupt it causes the translation mechanism to also update via dependency to change the ASCII character. This is a simple example but is sufficient to show how interrupts and dependency mechanisms work well together. The Cadence code for this was originally written in an older syntax[13], however, below is a small part of the driver updated to the new syntax. The example is for reading the scan-code and converting that to an ASCII character.

```
.devices ps2keyboard scancode is {
    if (@root interrupts 33) {
        @root io 0x60
    } else {
        0
    }
}

.devices ps2keyboard ascii is {
    if (.scancode < 0x80) {
        .asciimap (.scancode)
    } else {
        '\0'
    }
}
```

Listing 5.15: Keyboard driver example

A terminal output driver was also developed in the same system so that text

[13] An entirely different version of DOSTE, the one referred to in [Pope, 2007], which acted as an initial proof of concept for Cadence.

could be printed to the screen. This driver is considerably more complex than the above keyboard driver due to the huge number of observables and definitions that would be required if a pure OD-net Cadence approach (i.e. no agents) were to be taken. Each component of the screen, be that a pixel or character/attribute pair, would require a definition to describe its colour and many layers of abstractions would need to be built up on top of this to provide a useful interface. Such a pure Cadence approach is in practice too difficult. Instead agents need to be used to link parts together as multiple separate models could require output in what ultimately amounts to an unpredictable way. For this reason the old code behind this driver exploited an undesirable feature where definitions could have side-effects. In the most recent version of Cadence this capability has been removed and replaced by an alternative agent mechanism, however, the terminal driver has not been updated to use this new mechanism.

For character devices such as keyboards the observable dependency concept fits well with interrupts and IO ports. However, block devices and graphical output is not currently practical in Cadence without resorting to agency (not necessarily a bad thing). There is some possibility of having abstract definition templates that can be applied to millions of observables to deal with this problem but such features are far beyond the scope of this exploratory work[14].

5.3 Network Distribution for a Video Wall

Network distribution was another application for Cadence that has been put aside to focus on other aspects. The idea here was to have the underlying OD-net transparently shared between many machines so that hardware and software could be accessed as if it were local. The underlying architecture of Cadence supports extensions in a way that enables parts of a graph to be virtual or remote. By sending all events over a network instead of doing local processing it was possible to achieve the desired network

[14] As discussed in §2.1.3, Forms/3 does provide mechanisms for defining dynamic collections of cells in a lazy fashion. A similar approach could be taken to resolve the problem here with block devices.

Figure 5.9: Google earth running on the same video wall used for the Cadence work. Photo taken by Richard Cunningham

Figure 5.10: Stargate model running across 3 machines using XNet module

transparency at the lowest level. To demonstrate and develop this a video wall was used. This video wall consisted of 15 screens and 8 machines to drive them, each connected over an infiniband network (cf. figure 5.9.). A 9th machine was used as the server to control the others. Previously a software package called Chromium was used to distribute OpenGL rendering across the machines but this had many limitations in that it only supported a subset of OpenGL functionality at the time. It also sent polygon data over the network rather than any higher-level information about what was being drawn. Cadence takes a different approach. Each machine runs an instance of the Cadence environment but with a slightly different address space for nodes and is then connected to a master instance[15] on the master machine. At this point all machines can transparently observe the OD-net of the master machine so when a model is loaded each machine can observe it. It was then a simple task to customise each machine to draw a specific portion of the model from a particular view so that the result appears to be a single view over 15 screens. This customisation could be done live from the master machine since it only involved changing a few observables.

Performance was adequate despite the network code being poorly written. It had advantages over Chromium in that it could support all OpenGL features available on the graphics cards of the 15 machines. This included shaders. The model used to try

[15] There is nothing special about the master instance, it is the same program.

this out was the Stargate discussed previously. The Stargate model failed to run over Chromium but worked without alteration in this version of Cadence with all of the visual effects supported. Any model developed within Cadence could be adapted to run in this way and the video wall is just one example for distributed visualisation. Unfortunately the video wall was dismantled before any real data could be collected and analysed, although it was subsequently run across three machines in the lab to get the screen-shot in figure 5.10.

The network code is currently available in the XNet module. There is potential to use XNet for collaborative modelling which is important for scaling up plastic applications beyond the personal model. Although largely further work, §7.3.2 discusses how XNet was used in a lab session with students.

5.4 Kinesin Biological Model

One aspect of Cadence that can be further explored is the use of dynamical definitions. There are many models that make use of this concept, most of them for animation purposes. However there is one model that demonstrates the process like nature of these dependencies being applied for more than simply animation. The model is called Kinesin[16]. Kinesin is a motor protein operating within neurons as a means of axonal transport [Wilson, 2008]. They move neurotransmitter along microtubule tracks from the cell body to the synapses. The mechanism by which these motor proteins operate is not fully understood and so PhD researcher Richard Wilson has been attempting to construct a discrete dynamical system on a computer to learn about the process [Wilson, 2008]. His attempts have been using the C programming language to experimentally try out different theories he develops. It seemed plausible that constructing such a model in Cadence would provide him with a better platform for such experiments and so such a model was constructed.

The modelling process involved first creating a representation of the motor pro-

[16]The Kinesin model was developed in May 2008.

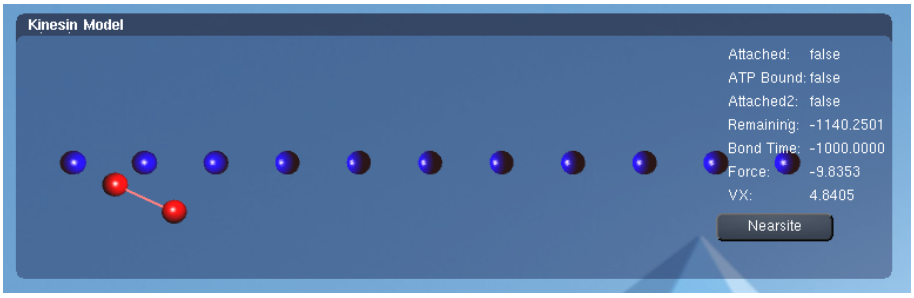

Figure 5.11: The Kinesin model during development

tein, consisting of two "heads" and a spring-like chain connecting them together. Once the basics were in place some dynamical definitions were added to simulate Brownian motion of these "heads". The precise sequence and timing of events is complex but important in understanding how it moves along the tubes. The process involves one head binding to the tube and releasing its ADP with ATP then binding to it causing the spring-like chain to partially attach to the top of the head. This encourages the other head to attach to the tube in front of the already bound head and the process repeats with this head. Meanwhile the first head hydrolyses ATP and ultimately this results in it detaching. By repeating this process the head walks along the tube. As our model progressed we were able to fine tune this sequence, based upon what is known about these various molecules and what was observed in the model, until we successfully and reliably got it walking. During the modelling process we did discover flaws in the original theory and realised that additional processes were involved that had not been considered[17].

Dynamical dependencies have been used in three ways within the Kinesin model. 1) For Brownian motion, 2) for the spring force calculations of the chains and 3) for the chemical reaction times of ATP. In each case it was a simple matter of entering the physical equations and seeing what happens. Being able to fine tune these whilst it is live was key to the rapid progress made in developing the model. The example in 5.16

[17] An additional delay was required to synchronise properly, something not originally accounted for. Wilson speculated that this delay was a chemical reaction previously not considered as important.

shows a velocity calculation using dynamical dependency:

```
vx is {
    if (.attached) {0.0}
    else {
        ..vx + (..chain (..fx) / (..mass)
        * (@root itime)) / (..dampx)
    }
}
```

Listing 5.16: Protein motion using dynamical definitions

As can be seen the head only has a velocity when it is free and then it is calculated from the standard $a = F/m$ for acceleration which is applied to change the velocity. The force has been calculated based upon the chain's spring effect and a random component for Brownian motion. In each case there are properties to control the strength of each effect which were adjusted experimentally to find what would make it walk. Once working values were obtained some explanation or comparison with reality was done to see if they were realistic.

In addition to demonstrating the use of dynamical dependencies the model also illustrates, to a limited degree, the benefits of cloning in a concrete interactive environment. The bond points that heads can attach to are constructed by cloning the first bond point which was created directly. Also, the second head of the protein was created by copying the first. Listing 5.17 gives part of the cloning code in the Kinesin model.

An observable called *i* has been added to the bond point which is used in the various definitions as an index so that each bond knows which one it is. When the bond point is cloned only the index observable *i* needs to be changed since the definitions will deal with all the other required changes. This is an example of a Cadence design pattern for indexed clones and occurs frequently in models. Without being able to clone in this fashion it would be necessary to either duplicate the code or somehow construct a procedural loop for generating objects, as is currently done in EDEN models of a similar

```
.bonds 0 = (new
    i = 0
    x is { -12.0 + (@axonworld bondspace * (.i) }
    y = -0.4
    next is { @axonworld kinesin bonds (.i + 1) }
    prev is { @axonworld kinesin bonds (.i - 1) }
    ...
)
.bonds 1 = (new union (.bonds 0) i = 1)
.bonds 2 = (new union (.bonds 0) i = 2)
...
```

Listing 5.17: Cloning of Kinesin bond points

nature (cf. problems B1 and B2 in §3.3.1).

It took considerably less time to reach this point with the model using Cadence than it had taken Wilson with his C program. This was due to the ability to observe and modify the process in gentle-slope and intuitive ways as it was running, whereas with C it involved stopping the program, changing the source, compiling and restarting. Our model was developed to more closely match the domain by using observables and definitions that had direct meaning rather than dealing with a more abstract machine based approach to state and behaviour. It was the direct, concrete and interactive nature of our tools that proved so beneficial in this kind of work and has shown how Cadence does apply the EUD principles of liveness, directness and gentle-slope (cf. §2.1.1).

5.5 Wii-fly Game and Presentation

Wii-fly is a game developed for and with the Warwick Game Design society at the University of Warwick[18]. The main purpose was to make a game that used wii remotes. Cadence and the associated game library provided a simple means of connecting wii

[18]Wii-fly was developed in February 2009 with the help of Sam Gynn.

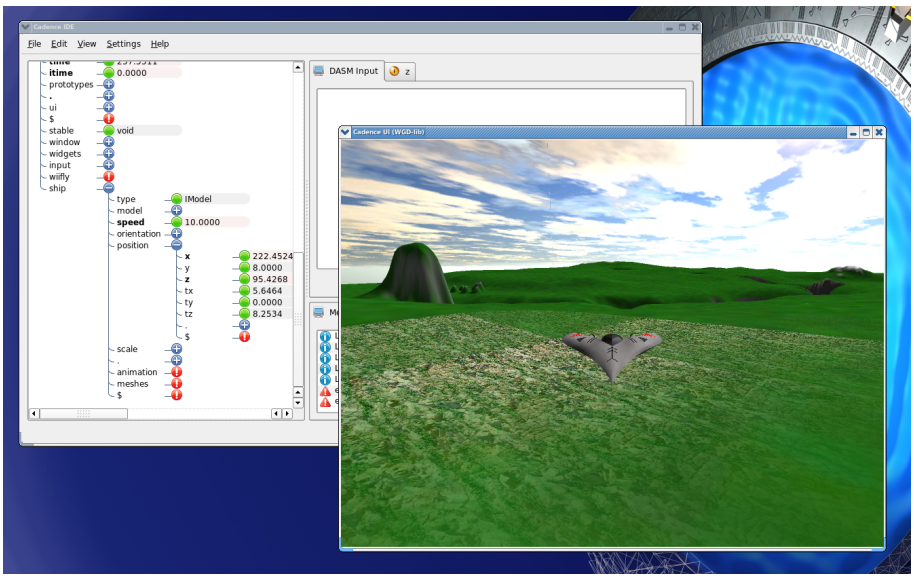

Figure 5.12: Early version of Wii-fly game with Cadence IDE

remote sensors to the controls of a "ship". The original and simple version of this game required only 161 lines of DASM script to create a world, a ship and connect the ship to the wii remote (cf. Appendix A). From this it was extended, adding menus and several different ships. Eventually some of the code was taken into a C++ agent but it still made use of Cadence.

What this game shows is how a simple module, which uses bluetooth to get Wii remote data, can be plugged into Cadence. This module provides a single agent that sends events into Cadence many times a second to update certain observables corresponding to buttons or accelerometer data. It also shows how Cadence can easily be used to connect this with other observables via various formulas in the form of definitions.

The player of the game controls a ship that flies over a landscape hunting down other players also flying ships. The game can work with up to 4 wii-remotes. During development there was a great deal of experimentation to get the ship behaviour correct as well as the gradual addition of new features. The response from the developers

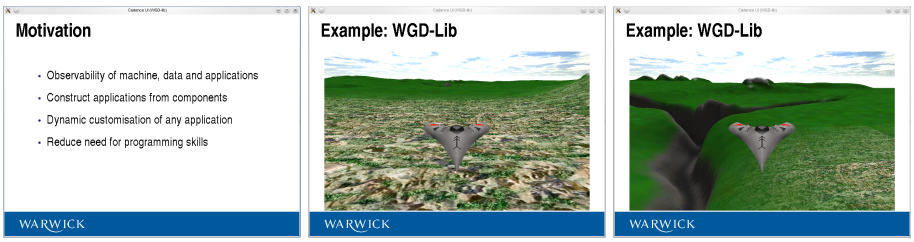

Figure 5.13: Cadence as a Presentation Environment

was mostly positive apart from some missing or incomplete features in Cadence which required a C++ component to work around them[19].

At around the same time a presentation was given on Cadence and the tool itself was used to give this presentation[20] (cf. figure 5.13). Individual slides were described using text objects and sprites in the game library and dependency was used to position them relative to each other on the screen. The purpose was to embed the wii-fly game into one of the slides to demonstrate the tools capabilities. Further, the wii-remote was also used to move from one slide to the next with the addition of a single dynamical dependency definition, shown in listing 5.18.

```
this widgets root children slide := {
    if (@wiimotes 0 buttons b) {
        ..slide next
    } else {
        ..slide
    }
};
```

Listing 5.18: Wii-remote to change slides

Perhaps of all the examples shown so far this is the closest to the vision of a plastic application, a model that looks to be developed traditionally but is open to

[19] At the time there was no support for anything other than C++ agency and agent action was required. This has since been resolved.

[20] The presentation was given in July 2009 as a PhD progress report.

change and experiment. Significantly it is the structured nature of the OD-net that has made it possible to combine the wii-fly game and presentation with ease, as well as allowing the slides to be developed by cloning from a template slide. The wii-fly game is embedded into a slide, as shown in listing 5.19, where first the wii-fly context is created, then the game is loaded and finally it is positioned within the slide. The presentation shows model reuse as well as showing the absence of observable aliasing problems (cf. problems A2 and B4 in §3.3.1). A presentation environment has been developed for *tkeden* [Harfield, 2007a], however this is less flexible and considerably more complex to use. The presentation in Cadence did not take much longer to put together than one in Powerpoint (or similar) would have taken.

```
@wiiwidget = (@slides 4 children);

%include "wiifly2008/wiifly_game.dasm";

@wiiwidget wiifly
    x = 40
    y = 100
    width is { @window width - 80 }
    height is {
        @window height - 150 - (@slide warwick height)
    };
```

Listing 5.19: Embedding Wii-fly into a slide

Chapter 6

Cadence Framework

This chapter looks to answer, with reference to the work of previous chapters and the subsequent chapter, the main questions posed in §1.4, namely: *how to enable a transition from construal to program by looking for specific improvements to Empirical Modelling tools and concepts*. With the conjecture that supporting such a transition will enable plastic applications. The *specific improvements* were identified in §3.4 and were prototyped in an environment called Cadence in chapter 4. To answer the question of whether a transition from construal to program is possible with Cadence and whether this intern enables plastic applications, it is necessary to first show how Cadence supports construals and how it supports the notion of program. The Cadence environment of chapter 4 is only a prototype but provides a practical basis for this discussion. The discussion needs to transcend the prototype of Cadence in chapter 4 and develop a Cadence framework that can look beyond the limitations of the implementation. The differences between the Cadence implementation and the Cadence framework highlighted here are issues for further work and so are briefly discussed in chapter 8.

To show how Cadence supports construals and programs it is necessary to give both an informal and formal semantics. Therefore, a provisional retrospective attempt is made to formalise Cadence in order to be more precise about its concepts. This attempt is by no means rigorous, but is suggestive of a possible approach that may be of use in

talking about programs. Once it has been shown how Cadence, in principle, supports construals and programs it is then possible to discuss how to migrate from one to the other (and back again) and how this gives the *plasticity* to applications. Chapter 7 follows this to demonstrate what is claimed here by looking at Cadence in relation to Empirical Modelling, with examples of both construals and programs.

6.1 The Concepts in Cadence

The Cadence framework in general involves a collection of concepts that are closely related to the Observable Dependency Agency (ODA) concepts of Empirical Modelling (cf. §2.2.2). Due to the changes suggested in §3.4 and implemented in chapter 4 there are now differences between the Cadence equivalent of ODA and the original EM ODA, the difference being largely in interpretation. It is necessary to be more precise about what the Cadence concepts are if it is going to be possible to identify how Cadence supports both construals and programs. A provisional and incomplete formal account is given here, however, it is sufficient for the aim of this thesis and what is lacking is taken up in chapter 8 as suggested further work (cf. §8.2.8).

6.1.1 Observables

The EM concept of observable was stated in §2.2.2 as: "*something which has a value or status to which an identity can be ascribed*". Whilst this definition remains true for Cadence there is a need to be precise about what is meant by "value" and "identity". The "something" is something from experience which has been observed and is considered (subjectively and provisionally at first) to be relevant to the modelling activity at hand. Due to the richness of experience it is not desirable to restrict this to particular types of thing, something discussed in §3.3.1.

Let $\mathcal{R}$ denote the set of all urelement[1] tokens[2] that represent something in

[1] Urelements are elements that do not themselves contain elements but are also not the empty set. Urelements are not sets.

[2] Corresponds to the OIDs in chapter 4.

experience (cf. §6.3.1). This is not a well-defined set in that experience is exceptionally broad. Figure 2.2 indicates that experience is unbounded and that there is a necessary gap between pure experience and construal. The identification of discrete tokens is already an abstraction but one that is required to make sense of the world and required to model that world on a finite discrete computer. They are tokens intended to represent the "terms" in experience that James identifies as atomic rather than relational in character [James, 1912/1996; Beynon and Pope, 2011] and is an "element that is directly experienced". However, James acknowledges that "experiences come on an enormous scale ... we have to abstract different groups of them, and handle these separately if we are to talk of them at all" [James, 1912/1996, 132]. How individual tokens in $\mathcal{R}$ relate to experience cannot be given formally and the tokens are not typed. The set of tokens may be finite[3], however, here it will be considered as being infinite since it will be useful for $\mathcal{R}$ to contain tokens for all integers $\mathbb{Z}$. The set $\mathcal{R}$ also contains tokens representing words, sentences and objects, such as an *orange*, which may not themselves be describable as sets. It is up to the observing agent to interpret these tokens in the "right" way; different agents may interpret tokens in different ways[4].

$$\mathcal{R} : \text{Set of all (urelement) tokens} \tag{6.1}$$

Agents may choose particular sets of tokens to correspond to, for example, existing mathematical objects such as the integers. $\mathcal{R}_{\mathbb{Z}}$ can be defined as in eq. 6.2 to be a collection of arbitrary tokens which are a subset of $\mathcal{R}$ but has the same cardinality as the set of integers $\mathbb{Z}$. In this way a notion of *type* can be developed on top of $\mathcal{R}$ as a standardised restriction imposed by agents (cf. §6.3.2).

[3] In practice the set $\mathcal{R}$ is finite

[4] A public interpretation of particular tokens relates to the migration from construal to program (cf. §6.3.2).

$$\mathcal{R}_{\mathbb{Z}} = \{\ldots, r_{-1}, r_0, r_1, \ldots\} \tag{6.2}$$

$$\mathcal{R}_{\mathbb{Z}} \subset \mathcal{R} \tag{6.3}$$

$$|\mathcal{R}_{\mathbb{Z}}| = |\mathbb{Z}| \tag{6.4}$$

There also must then exist (in the "mind" of the agent) a mapping from the integers to these tokens, described by the function l in eq. 6.5, and an inverse mapping from tokens back to the integers. l is a bijective function.

$$l_{\mathbb{Z}} : \quad \mathbb{Z} \rightarrow \mathcal{R}_{\mathbb{Z}} \tag{6.5}$$

$$l_{\mathbb{Z}}^{-1} : \quad \mathcal{R}_{\mathbb{Z}} \rightarrow \mathbb{Z} \tag{6.6}$$

Observables are distinguished from each other in experience to become the atomic "terms" of experience represented by tokens in $\mathcal{R}$. Relating this back to the EM definition, an observable is identified by its token and its value or status is attributed to it by an agent mapping from token to a particular experience. This interpretation of observable is somewhat different to that used in EDEN where observables resemble program variables[5]. The key difference is that observables in Cadence cannot change value in the same way as they can in EDEN[6].

Observables in EM tools to date have been identified using syntactically restricted names which is problematic (cf. naming issues in §3.3.1). It is not always possible to name an observable appropriately. Within the Cadence prototype implementation some tokens[7] are given names using provided mapping functions like $l_{\mathbb{Z}}$. All primitive C++ data types are given such mappings and users may also associate names with particular tokens (usually randomly chosen tokens) so that they can share standard interpretations

[5] Observables resemble program variables only in that they can have a value assigned to them.

[6] To change value in Cadence means an agent must change its interpretation of what experience the token represents.

[7] Tokens are not themselves names and are never intended to be known directly by an agent. This is also important for Capability-based security.

between agents[8]. Except for these special tokens that are named, the remaining tokens can only be "found" through their (structural) relationship to other already known tokens[9]. A common example of using unnamed tokens is given below in listing 6.1.

```
. a = (new);
```

Listing 6.1: Observable identification in DASM

The simple DASM example in listing 6.1 could be interpreted in existing EDEN terms as describing an observable called 'a' with an object value. With a Cadence interpretation there are three observables given in the example, one giving the context, one with an associated name 'a' and one that corresponds to some unknown token generated by '**new**' (e.g. <1:0:1:3248>). The context is some larger containing experience and the named observable corresponds to some experienced (structural) relation[10]. The observable on the right-hand-side cannot be identified directly with a name (it has not been given a name), instead it is identified by the structural relationship with the other two observables.

This approach to identifying observables is similar to that used by Subtext (cf. §2.1.3) which is trying to move away from paper-centric approaches including the need to give unique names to things. Instead a direct approach to finding variables is taken using an interactive interface. Names are an abstraction so by not requiring names the artefact can become more concrete in the sense that not everything in experience can be given a name unless abstracted. The current Cadence implementation relies rather heavily on the DASM notation which makes it difficult to move away from naming observables. A better, more direct, interface would be vital in fully escaping the need

[8] In a distributed version of Cadence it is necessary to use a global set of such mapping functions so that each agent can interpret the OD-nets on other machines. Otherwise what one agent thinks of as a token for the number 5 might be considered to be the number 6 on a different machine. This problem is a real one.

[9] An agent will always have at least one known token initially.

[10] Further work is needed on explaining what it means to say that a relation is an observable experience (in an abstract sense) but it relates to William James who argues that conjunctive relations are experienced.

for textual names, something explored in §8.2.2 as future work.

6.1.2 Relationships and Dependency

Relationships play a vital role in Cadence for the same reasons they do in Empirical Modelling (cf. §2.2.2) as well as for identifying observables, as mentioned in §6.1.1. An EM definition for dependency is: "*a relationship between observables that - in the view of a state-changing agent - expresses how changes to observables are indivisibly linked in change*". There is, however, another form of "experientially mediated association" which is not made explicit in the definition of dependency given above. Structural relations between observables are not dependencies but are the *subject* of dependencies in that the "change" being referred to is in fact, in Cadence, a structural change[11]. This is a consequence of observables being tokens and not variables, they have only an identity inside the OD-net, their value is external. Further, the notion of dependency can be split into two distinct kinds of relationship when time is considered, latent and dynamical. Whilst the EM definition of dependency can account for both kinds, the EDEN tool only accounts for latent relations (cf. §3.3.2). This section explores and attempts to provisionally formalise the three distinct kinds of relation along with the notions of meta-structure and meta-dependency.

Structural Relations

The first kind of relationship in Cadence, which is not well supported in EM to date (cf. §3.3.1), is a structural relation. A structural relation describes how observables spatially (or logically) relate to each other and corresponds to the graph in Cadence. Structural relations are fundamentally important in Cadence as they are the primary means of identifying observables and they provide the foundations for the other types of relation, as well as being the basis for computation (cf. §4.3.1). It is being argued here that

[11] EM does not exclude the idea of structural relations and there is some limited support in the EDEN tool for structure using lists. However, this is so limited and poorly dealt with as to be essentially irrelevant here.

structure is a fundamental concept largely neglected in EM to date (cf. §6.3.1). The most basic form of algebraic structure is a *magma* (cf. §3.4.1) which is an appropriate choice for a highly flexible domain agnostic representation for structure, and is given in eq. 6.7 using the set of tokens $\mathcal{R}$ defined above in eq. 6.1.

$$\phi : \mathcal{R} \times \mathcal{R} \to \mathcal{R} \tag{6.7}$$

$$\phi(r_x, r_y) = r_z \tag{6.8}$$

Structure is obtained from ϕ by composition of ϕ with itself, possible because of the *closure* of ϕ over $\mathcal{R}$. Using the graph terminology introduced in §4.1, r_x in eq. 6.8 is a node, r_y is the edge and r_z is another node. A graph interpretation of ϕ is only one possible interpretation which has proven useful conceptually and will continue to be used here[12]. However, the formal description of structural relations is independent of a graph interpretation. Eq. 6.9 corresponds to the DASM query in listing 4.1 and the graph in figure 4.4.

$$\phi(\phi(\phi(this, r_{\mathrm{x}}), r_{\mathrm{y}}), r_{\mathrm{z}}) \tag{6.9}$$

Structural relations are not dependencies as defined by EM, although there may be some deeper notion of dependency in that the meaning of any particular observable[13] may *depend* upon its structural relationship to other observables (for example, the concept of *parent* is a kind of structural relationship that could change causing the parent to no longer be a parent in the eyes of an observer). This kind of dependence is an "experientially mediated association" that is not itself a dependency, however, dependencies may exist upon it through the use of dependency relations (latent or dynamical). In fact, in Cadence, all dependency comes back to dependence upon structural relations as it is only the structure of (definition of) ϕ that can change.

[12] Other interpretations include an object hierarchy, a lookup table or a binary function.

[13] As in the way it is observed and interpreted by agents

Whilst structural relations can be enumerated as in eq. 6.8, they can also be described abstractly as meta-structural relations (or meta-structure). Meta-structure is not concrete (infinite rather than finite structure) but instead a generic description that makes use of one or more mathematical variables. The simplest meta-structural description involves a single variable, as shown in eqs. 6.10 and 6.11. The symbol r_x represents an arbitrary token in $\mathcal{R}$ and α is a variable for all possible tokens (unless otherwise restricted).

$$\phi(\alpha, r_x) = \alpha \tag{6.10}$$

$$\phi(r_x, \alpha) = \alpha \tag{6.11}$$

Eq. 6.10 states, in graph terms, that for all nodes there is a directed edge labelled r_x that points back to the same node. Eq. 6.11 defines all edges of a node r_x to point to the same node (token) as is used to label the edge. Meta-structure can also be used to describe the arithmetic operators, as shown in eq. 6.12 which gives the addition operator. Multiple variables are required for the addition operator, as is a token r_+ which is used to represent the addition operator.

$$\begin{aligned} \phi(\alpha, r_+) &= \beta \quad \text{if } \alpha \in \mathcal{R}_{\mathbb{Z}} \\ \phi(\beta, \gamma) &= l_{\mathbb{Z}}(l_{\mathbb{Z}}^{-1}(\alpha) + l_{\mathbb{Z}}^{-1}(\gamma)) \quad \text{if } \gamma \in \mathcal{R}_{\mathbb{Z}} \end{aligned} \tag{6.12}$$

The conversion of a binary operator such as addition into *magma* form can be generalised by first transforming it into a generic ternary function f where the first parameter identifies the desired operation. This function f (eq. 6.13) can then be partially decomposed into f' and f'' (eqs. 6.14 and 6.15), both of which are of the same form as ϕ when $\mathcal{R}_{A,B,C} \subset \mathcal{R}$.

$$f: \quad \mathcal{R}_A \times \mathcal{R}_B \times \mathcal{R}_B \to \mathcal{R}_B \tag{6.13}$$

$$f': \quad \mathcal{R}_A \times \mathcal{R}_B \to \mathcal{R}_C \tag{6.14}$$

$$f'': \quad \mathcal{R}_C \times \mathcal{R}_B \to \mathcal{R}_B \tag{6.15}$$

$$f(r_x, r_y, r_z) = f''(f'(r_x, r_y), r_z) \tag{6.16}$$

The default initial definition of ϕ in a Cadence implementation is shown in eq. 6.17. Every edge from every node points to the *null* node[14], it is then up to agents to incrementally modify the definition of ϕ to describe more interesting relationships. The modification of ϕ must always result in it being well-defined which relates to the indivisibility of change that is so important for experientially mediated associations in EM[15]. It is interesting to note that using this approach all possible observables already exist in the system, as do all edges from each node (in that they point to *null*) and that it is only the relationships between observables which are changed to provide meaning when observed, hence why structural relations are the foundation of Cadence.

$$\phi(\alpha, \beta) = r_{\text{null}} \quad \forall \alpha, \beta \in \mathcal{R} \tag{6.17}$$

Latent Relations

Latent relations are a kind of dependency and are the relationships supported by EDEN (cf. §2.2.3). Dependency relations define, in an algebraic sense in Cadence, the structural relations of observables as being some expression involving other structural relations and other observables. In other words it involves defining ϕ recursively by using ϕ on the rhs of its own definition, as shown in eq. 6.18. The structural relation being defined is then dependent upon the observables and their relations used in the expression. Such relationships rightly need to be indivisibly maintained when change occurs for all

[14] The *null* node is also in $\mathcal{R}$

[15] This will also have consequences for concurrency in a multi-agent model.

structural relations as a whole to remain coherent with respect to their dependency relations, although this is implicit here if ϕ remains a well-defined function. A latent relation is a dependency relation where it is invariant with respect to time as perceived by an agent observer. In other words, such relationships only become apparent when agents make a change (hence the use of *latent*), in contrast to dynamical relations (cf. §6.1.2). In the Cadence prototype, latent relations are described by latent definitions and passive dependency maintenance (cf. §4.3.2) is used to indivisibly maintain coherence.

$$\phi(this, r_{\mathrm{a}}) = \phi(\phi(\phi(\phi(\phi(this, r_{\mathrm{b}}), r_{\mathrm{c}}), r_{\mathrm{d}}), r_{\mathrm{e}}), r_{\mathrm{f}}) \tag{6.18}$$

Given the definition of ϕ in eq. 6.7 it is possible to write the latent definition given as the DASM statement in listing 4.19 as shown in equation 6.18. Part of ϕ is being defined here in terms of itself, but not in a cyclic fashion which is of course meaningless. As with structural relations there are also meta-dependency definitions which abstractly describe general dependency relationships. Eqs. 6.19 to 6.22 give the most basic examples of meta-dependencies.

$$\phi(\alpha, r_x) = \phi(\alpha, r_y) \tag{6.19}$$

$$\phi(\alpha, r_x) = \phi(r_y, \alpha) \tag{6.20}$$

$$\phi(r_x, \alpha) = \phi(\alpha, r_y) \tag{6.21}$$

$$\phi(r_x, \alpha) = \phi(r_y, \alpha) \tag{6.22}$$

Eq. 6.22, for example, states that all edges from some node r_x point to the same node as the corresponding edge from node r_y. This is an example of how inheritance may be described using meta-dependency because now the structures for r_x and r_y appear to be the same except that r_x could be further defined to override the definition in eq. 6.22 for specific cases.

$$\phi(r_x, \alpha) = \phi(\phi(\alpha, r_\times), \alpha) \tag{6.23}$$

The equation in 6.23 corresponds to the example given in listing 4.27 which describes a generic square operation (there is no reference to any particular kind of object such as integers). The Cadence prototype only has basic support for meta-dependency as it only allows for a single variable[16].

The potential use of meta-relations is huge but largely unexplored in this work (cf. further work in §8.2.6). The problem is in relating the use of meta-relations to the development of construals which are concrete. Meta-relations are intrinsically unobservable (abstract) until used as they require values for the variables. Although the generic definitions of §4.3.4 are the meta-dependencies being discussed here, there is another kind of generic definition which may be of some use. Sometimes it may be useful for agents to have parameterised template definitions to help with model construction. These template definitions are not a part of the OD-net itself as they would be instantiated by agents before being added to ϕ. They form an external library of definition templates.

Dynamical Relations

Dynamical relations are another form of dependency relation that describes how structural relations change over time (cf. §4.3.3) and so is time variant from an agents perspective and *appears* to change without agent interaction (cf. §6.1.3). This kind of relationship is not well supported by EDEN[17] but is not excluded from the definition of dependency used by Empirical Modelling. To account for time a set of ϕ functions can be associated with a natural number corresponding to a discrete instant in time. This is done with Φ as in eq. 6.24.

[16] The built-in arithmetic operators are also conceptually meta-relations that have multiple variables, however, they are meta-structural

[17] A new observable could be created in EDEN for each discrete instant in time and then have agents change which observable they observe depending upon the present instant (cf. §7.1).

$$\Phi : \mathbb{N} \to [\mathcal{R} \times \mathcal{R} \to \mathcal{R}] \tag{6.24}$$

$$\Phi(t) = \phi_t \tag{6.25}$$

The shorthand ϕ_t will be used for the rest of this section. The initial default definition of ϕ now needs to be modified to the following:

$$\phi_t(\alpha, \beta) = \begin{cases} \phi_{t-1}(\alpha, \beta) & \text{if } t > 0 \\ r_{\mathsf{null}} \end{cases} \tag{6.26}$$

The example definition in eq. 6.27 corresponds to the DASM example in listing 4.24 which continually changes a structural relation between observables such that, when interpreted by an agent, it appears to be counting up by one each instant[18]. A now variable has been included which relates to the modifying agents concept of the present instant at the time the definition was made.

$$\phi_t(this, r_{\mathsf{a}}) = \begin{cases} \phi_{t-1}(\phi_{t-1}(\phi_{t-1}(this, r_{\mathsf{a}}), r_+), r_1) & \text{if } t > now \\ r_0 & \text{if } t = now \\ \dots \end{cases} \tag{6.27}$$

Unlike the Cadence implementation in chapter 4, the description of Cadence given in 6.24 allows definitions to refer more than just one time step into the past or future. This limitation in the implementation is discussed in §8.2.4 as further work. Whilst references to the future are conceptually allowed, if mixed with references to the past they will readily result in cyclic definitions (across time) which cannot be permitted[19]. For this reason it is important to either always refer to the future or, more likely, always refer to the past. The latent relations discussed above are a special case of these dynamical relations where the time component t does not vary between the

[18] As in the tokens when interpreted as integers increase over time.

[19] Only in situations where references to past and future are disjunctive will it work.

left-hand-side and right-hand-side of the definition[20]. It is useful, however, to separate the two concepts[21].

6.1.3 Agency

In Cadence there is no difference in the role of agency from that found in the existing EM framework. The definition in EM is, therefore, valid for Cadence: "*an agent is projected on to the referent as something that can change the state of the model in some way by manipulating observables and relationships*". In other words, agents manipulate and observe the OD-net (ϕ_t) in ways similar to how they might manipulate and observe the referent. This thesis has not explored the nature of agency in Cadence in depth. What is important to acknowledge is the nature of "change" now that dynamical relations have been identified, and what the different roles of agents may be.

Whereas dependency is a part of the model of state-as-experienced and describes indivisible relationships, agency is used for experimentation with that state and for boundary situations between what is and is not in scope. Agency could be used instead of dependency throughout the model, however, this would result in a traditional imperative environment which, it is argued by EM, is not as appropriate for construals. The benefits of using dependency would be lost if agency was not confined to the boundaries where dependency is not possible[22].

Types of Change

It may be appropriate in some circumstances to associate two kinds of change with Cadence, however, this depends upon the context for observation. The two types of change might be, from an agents limited perspective, as follows:

1. Internal - mediated by deterministic dynamical relations.

[20] The EDEN tools observables and dependencies are therefore a subset of the ϕ relation.

[21] The implementation of these relationships is different in the Cadence prototype (cf. §4.4.2).

[22] Dependency cannot be used at the boundaries because it would require knowledge of observables and dependencies outside of scope, potentially resulting in the need to model the entire universe. A line needs to be drawn where agency is used instead.

2. External - enacted by non-deterministic actor agents.

Internal change is "illusory" since the definition of Φ is not changing. The illusion of change is the result of an observer's own motion through time and their limited view point which restricts them to only observing ϕ_{now}. An observer would also experience change if they were to shift their focus to different observables at the same point in time.

By restricting agent observation to ϕ_{now} and continually changing the now variable (hence continually changing ϕ_{now}), it is possible to simulate change and therefore animate the artefact in a deterministic manner[23]. The Cadence implementation in chapter 4 only allows agents to observe ϕ_{now} and also imposes restrictions upon possible modifications to the definition of ϕ (cf. discussion of dynamical relations in §6.1.2). These restrictions on observation and modification are potentially serious limitations since it may be appropriate for some agents (e.g. the human modeller) to observe Φ and have access to history[24], whilst choosing to restrict the capabilities of other agents to ϕ_{now}. Other restrictions to agent observation in addition to ϕ_{now} may also be appropriate, especially when moving from construal to program. Such deficiencies are identified in §8.2.5 as future work.

External change is caused by external modification of the definition of ϕ_{now} by actor agents (either internal or external actors) and as a result is non-deterministic from the point-of-view of other agents within Cadence. Agents cannot, no matter how unrestricted their observational capabilities of the OD-net, predict changes caused by other agents[25].

[23] Although it may appear to be non-deterministic to an agent with restricted capabilities for observation.

[24] Both to observe and manipulate the past and see consequences propagate automatically through time by dependency maintenance. Relates to the *steering* supported by Forms/3 (cf. §2.1.3).

[25] Unless they could directly communicate with each other by means other than the OD-net. Such situations are not considered to be a part of the Cadence framework but are also not excluded.

Role of Agency

One of the roles of agents in EM is to enable experimentation through interaction and observation of an artefact (cf. §2.2.2). It is possible, however, to identify four different kinds of agent in both Cadence and EM based upon the role they play in a model. These four kinds are grouped into two categories:

- Mediators: those agents which do not act autonomously but mediate between the artefact (OD-net) and some other agent.
  - Observers: agents which interpret parts of the artefact in specific ways to translate it into something another agent can then observe.
  - Manipulators: an interface through which other agents may make specific and potentially complex manipulations of the artefact.
- Actors: these agents are deemed to be responsible for some manipulation or are the ones making an observation. Typically actors will both observe and manipulate an artefact, either directly or via mediator agents.
  - Internal: agents which are automated and act autonomously within the Cadence environment.
  - External: agents outside of the computer, which will include the human modeller(s) and other sources of input/output.

These classifications of kinds of agent are perhaps overly simplistic and there is much scope for looking at this further, especially in relating agents to the restrictions identified above such as ϕ_{now}. These issues will be discussed in chapter 8.

6.2 Development Process

Empirical Modelling [Sun, 1999] and Cadence are both fundamentally amethodical[26]. There are no formal procedures or sets of methods by which a model can be developed

[26] Amethodical, as defined in [Truex et al., 2000].

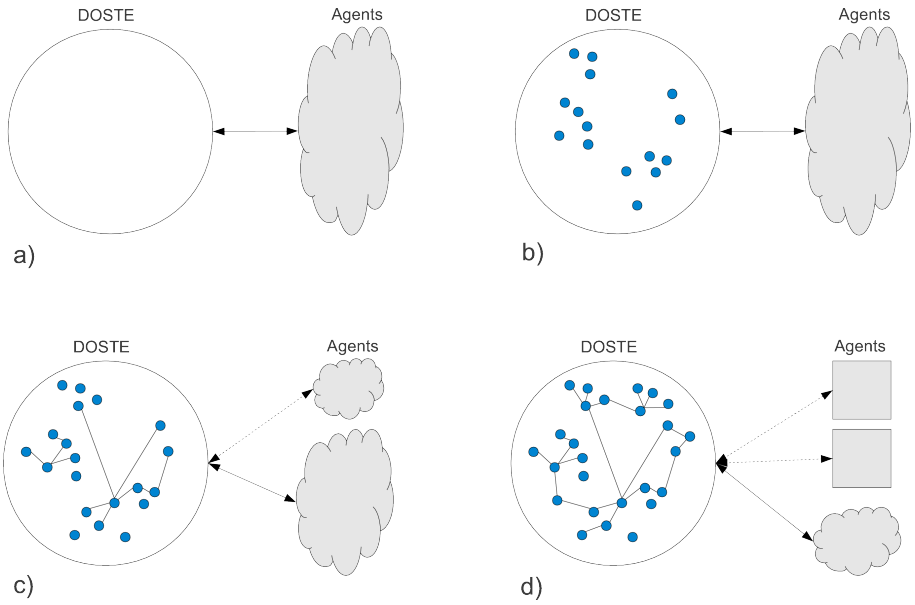

Figure 6.1: Evolution of a Cadence model. a) shows the initially empty environment. b) observables identified. c) first dependencies introduced and agents take form. d) extensive addition of dependencies and observables with well defined agents.

as each model is situated in a unique context and so must be developed in a unique way. This is the nature of plastic applications as outlined in the introduction. It is, however, possible to give guidelines and concepts which help with the development process and hopefully prevent the modeller from becoming "puzzled" by not knowing how to proceed [Sun, 1999, p.71]. The Cadence development process is similar to that of EM, with the EM process being discussed by Sun in his PhD thesis [Sun, 1999], which has been illustrated in figures 2.2, 2.6 and 2.10.

The primary concepts in Cadence were given above in §6.1. Figure 6.1 shows how a model might evolve by gradually identifying observables, relationships and agents. The models of the previous chapter, such as Stargate and Kinesin (cf. §§5.1.3 and 5.4), illustrate particular examples of a model development process in Cadence; further examples will be given in the next chapter. There are four key differences between the Cadence development process and the current EM development process which will

substantially alter the way Cadence models develop, especially as they grow in size and move towards larger programs.

1. Support for component separation by using structural relationships to delineate objects. Each component can become a focus of observation and be refined independently of the rest of the model (cf. §3.4.1). Such component separation enables reductionist approaches to be taken if appropriate and allows for decomposable test units (cf. §2.1.4). Component separation played a significant role in the development of the Stargate model, as illustrated in §5.1.1 with the bloom effect.

2. Abstraction of collections of observables and relationships into generic prototypes that can be reused both within and between models. Component reuse would enable a rich library of such components to be developed to help with scaling up the models. An example is the game library (cf. §4.4.5) which provides prototypes for interface elements such as buttons. Each new model could reuse existing well refined models (cf. §5.5) and so does not need to start from the ground up each time [Sun, 1999, p.71]. Abstraction into generic prototypes should be encouraged where appropriate.

3. The introduction of dynamical relationships allows more to be moved from agents into the artefact itself. Consequently, the role of agents has changed and so through the refinement process the agents will shrink down to the boundaries[27] rather than remain playing a central animating role. In general as the program is refined the agents should be reduced to their simplest forms.

4. Removing the focus on definitive scripts and instead interacting directly with the OD-net changes the way models are managed. The vision in Cadence is for the OD-net to be persistent and distributed (cf. §8.2.4). Managing and separating

[27] The boundaries refer to the scope of the model, the boundary between what is a part of the model and what is not.

models now needs to take advantage of structural relationships rather than script files. Additionally, if OD-net history is supported in the future, then this would act as a form of version control.

6.3 Supporting Plastic Applications

This thesis has taken the approach that plastic applications can be supported by allowing individual human modellers to start out by developing construals and refining these sufficiently far that they become "programs". Having described the Cadence framework in this chapter it is now possible to discuss the extent to which this Cadence framework supports the migration.

6.3.1 Supporting Personal Construals

Earlier sections of this thesis have already introduced the notion of construal (cf. §2.2.1), how construals may be supported by computers (cf. §2.2.2 and §2.2.3) and how this relates to plastic applications (cf. §3.1). The *dimensions of refinement* identified in §3.2 give the characteristics of construals: they are personal, subjective, provisional and specific. The first question to ask is: *does Cadence support construals*? A simple way to answer this is to 1) develop construals using Cadence to demonstrate empirically that it does (cf. kinesin model in §5.4) and 2) by relating Cadence to Empirical Modelling which already has a great deal to say about supporting construals. Both points are explored further in chapter 7. For now it is easy to claim that because Cadence is based upon EM concepts, and in fact extends them, that it does support the development of construals, at least as much as EM does.

A way to measure how well a software environment such as Cadence supports construals is to look at how well it deals with each of the following issues which relate to the desirable characteristics of construals:

- Variety of ontologies[28] supported so that personal and subjective decisions can be made about which approach to take.
- Connecting different ontologies together - if many are used - to provide a coherent artefact.
- Reducing problems with making ontological commitments by allowing transitions from one approach to another and so keeping the construal provisional.
- Support for experimentation with the construal to help identify what ontological decisions to make. This can be split into two:
  - Freedom to observe and interpret the artefact subjectively.
  - Freedom to interact with the artefact to gain understanding of it.

The remainder of this section will look at how the Cadence tool and framework measure up against these criteria. It is important to remember that the focus is not on better support for construals specifically, at least not beyond what EM can already do, but on moving from construal to program. How this objective has impacted upon support for construals does, however, need to be taken into account and it turns out to be significant[29].

Variety of Ontologies

The primary means of supporting different representations for different purposes in Empirical Modelling is the use of definitive notations. Notations exist that are suited to specific domains such as Scout for window management (cf. §2.2.3). The problems with using definitive notations were identified in §3.3.3. One such problem is that each do-

[28] The use of "ontologies" here is pragmatic, as often used in computer science, rather than being taken too philosophically.

[29] Initially it was not an objective to improve support for construals. Any improvements in support for construals that resulted were somewhat unexpected consequences of reinterpretations of existing EM concepts.

main requires a unique notation and that this is expensive. Without notational flexibility the construal must be framed in terms of the pre-existing notations.

Whilst Cadence could support definitive notations as mediator agents to provide the same functionality as EDEN, it is also possible to represent the structures and operations present in these notations directly in the Cadence OD-net. An example is given in listing 6.2 of a possible translation of the Scout script given earlier in listing 3.3. How exactly a Scout script is converted is for the modeller to decide due to the range of possible translations. It is clear when comparing the two versions (listings 3.3 and 6.2) that they look similar, and certainly the Cadence form is an improvement on the Eden translation in listing 3.4.

The argument here is that the Cadence OD-net (i.e. ϕ) is capable of directly supporting the Scout ontology. The same is also true for the other notations by a similar translation. Due to the computational nature of ϕ when traversed by agents, the notation specific operations can also be translated into the OD-net[30]. As a consequence there is little need for the use of definitive notations at the construal stage of model development, and this removes the problem of notation inflexibility (cf. issue E3 in table 3.2) which in turn allows for a greater variety of ontologies than EM currently supports with EDEN. Subject to implementation support for meta-structure, ϕ is sufficiently generic and unrestricted to represent many kinds of structural and other relationships as well as operations.

Connecting Ontologies

A problem identified back in §3.3.3 (problem E1), and which is not specific to EM (cf. meta-domain language in §2.1.4), is how to communicate between distinct kinds of representation. This is an active area of research, particularly in the flexible modelling tools community where they are attempting to bridge between different kinds of representation, mixing formal and informal together. An example is BITKit (Business

[30] Although the current implementation of Cadence is limited in its support for generic definitions (cf. §4.3.4)

```
.A1 = (new
    type = TEXT
    frame = (new
        x1 = 50
        y1 = 50
        x2 is { .x1 + (@gridsquare width c) }
        y2 is { .y1 + (@gridsquare height r) }
    )
    font = (new union (@prototypes font))
    bgcolor = (new r=1.0 g=1.0 b=1.0)
    fgcolor = (new r=0.0 g=0.0 b=0.0)
    bdcolor = (new r=0.0 g=0.0 b=0.0)
    border = 2
    relief = DEFAULT
    alignment = CENTRE
    sensitive = (new
        ON = true
        ENTER = true
        LEAVE = true
    )
    string is { @root a1 }
);
```

Listing 6.2: SCOUT window in DASM

Insight Toolkit) [Desmond et al., 2010] which is a smart office tool that enables an underlying model to be developed that can link together spreadsheets, presentations and documents through a model of the information rather than relying on a manual translation process. Such underlying representations need to be "domain agnostic" and "highly flexible and capable of evolving and refining an arbitrary meta-model" [Desmond et al., 2010]. Similarly, work by James Douglass at Boeing [Douglass, 2010] is looking at a "language of languages" to enable different views and means of interaction with what is fundamentally the same model[31].

The approach often taken, and the one taken here, is to find a generic foundational representation to which all others can be translated. The problem with EDEN is that the foundational Eden language is not up to the task (cf. §3.3). As already shown using listing 6.2, the foundational representation in Cadence (the ϕ function) is capable of appropriately supporting translations from other notations. By being able to work with the flexible and unrestricted Cadence representation of state there is substantially less difficulty in inter-representation communication than with Eden. This point is revisited in §7.1.1. Consequently it can be argued that Cadence is better (than EDEN) at connecting together different ontologies and so is better at supporting diverse construals.

Dealing with Commitments

Abstraction is required to simplify the world which is far too rich for us to reason about, even with the help of computers. Knowing what is important for any particular purpose is an important and difficult process that comes before any more formal approaches can be taken. Identifying what observables and relationships are important is a vital (perhaps the most vital) part of the process of developing construals and is highly context dependent as well as somewhat subjective. Traditionally this identification process would be the software design process which, although utilising various modelling tools, is usually an off-line process. Although the Cadence OD-net does support richer ob-

[31] Related to the "modes of observation" in Empirical Modelling

servable and relationship possibilities, it still needs to account for the possibility that previously identified relationships and observables may change as what is and is not important becomes clearer to the modeller.

The process of radically changing relationships and observables (on a large scale) is to be called *refactoring* because of its similarity to a process of the same name used in traditional programming[32]. Refactoring is required because, despite Cadence being exceptionally flexible, the modeller will make particular ontological commitments in a model and these may turn out to be inappropriate. This is indicative of the *provisional* nature of construals. The ability to refactor a construal is what really makes it a construal and is important for extreme plasticity.

In traditional object-oriented software development, designs are split into modules, components and objects with encapsulation being important. This is done in an attempt to hide details and reduce dependencies between components so that changes do not propagate throughout the software but are kept local. Localised change is easier to manage. Aspect-Oriented Programming attempts to further reduce cross-cutting concerns between components. Service-Oriented Architectures and dependency injection are also trying to reduce built-in dependency. These technologies allow for some degree of refactoring by minimising change propagation.

Instead of removing dependency, Cadence and EM take the opposite approach and embrace dependency to automatically propagate change. Through the use of dependency definitions like those used in the Stargate model (cf. §5.1) to provide short-cuts to shader variables (cf. listing 5.7) it is possible to create different but connected views of essentially the same thing. To a limited degree this allows for apparent refactoring which may in many cases be adequate. Also, due to the flexibility of agent interpretations a degree of refactoring can be done without any real underlying change to the OD-net.

The kinds of refactoring and change that are currently supported by the traditional technologies are also possible with Cadence. For example the use of objects in

[32] Refactoring programs involves changing the structure or design, without necessarily changing the functionality.

Cadence allows for modularisation and dependency injection (cf. use of game library in §5.1.1 Stargate model). More radical refactoring, where one or more objects need to be transformed into entirely different objects, currently remains a rather manual process within Cadence, where individual observables and definitions need to be rewritten. In fact, because of various limitations with the prototype tool (cf. §8.2), this is done in script files off-line rather than on-line as is intended. These limitations could be overcome by providing more sophisticated manipulator agents. Traditional software development approaches will also suffer from similar problems if such radical changes are necessary[33]. The use of dependency maintenance and support for context switching does help with this process by taking some of the burden of change away from the modeller, without requiring the modeller to explicitly design for change. The advantage of embracing dependency rather than removing it is that unexpected changes can be better dealt with automatically, but also it is easier to debug and trace problems using the computer rather than through off-line analysis. Radical redesign can become, and should be, a live experimental process.

Cadence has embraced ideas from the software industry on how to manage change and has integrated these approaches with the Empirical Modelling framework. The result is that Cadence is better able to support refactoring than EDEN. However, there is much room from improvement in supporting radical refactoring, discussed as further work in chapter 8.

6.3.2 Supporting Public Programs

The characteristics of a program were identified in §3.2 with the dimensions of refinement. A program is public, objective, assured and generic. The intention is that what was once a construal is now an artefact that can be publicly communicated, understood and used. No ambiguity over meaning and interpretation can remain. The artefact must also reliably reflect what it is intended to represent but also be abstract and

[33]Sometimes requiring a complete or extensive rewrite of the software if the change occurred after the implementation had begun.

generic enough to apply over a broader range of situations than the original concrete example used to develop it.

As with support for construals, one way to show that Cadence supports programs is to develop such programs. Examples of programs (albeit small ones) can be found in the following chapter (cf. §7.3.3). There are key ways, however, in which Cadence is better able to support programs than EDEN:

- By using agents only for boundary situations (i.e. input and output) the rest of the model can be migrated into the OD-net which has a formal basis not dissimilar to functional languages (cf. §6.1). Removing excess use of agency for internal state maintenance allows the program to be more easily formalised due to its deterministic nature. It is the lack of support for custom agents in the Cadence prototype that has meant that entire models needed to be described directly with the OD-net. All of the models in chapter 5 illustrate this. Whilst the lack of custom agents is problematic for supporting construals it has shown how Cadence is able to function without internal agency, something which EDEN is not able to do (cf. §3.3.2). There is much potential for automatic just-in-time style optimisations (similar to Java and Self) and concurrent implementation of the Cadence OD-net (cf. §§4.4.2 and 8.2.7).

- Generic prototyping design patterns, cloning and the ability to switch groups of observables allows types and components to be abstracted (cf. Kinesin bond prototype in listing 5.17 and button prototype in appendix A). The program can scale up and adapt to different scenarios more readily by using these generic adaptable examples, whereas in EDEN it would involve extensive use of agency (to generate definitive scripts) to make even simple adaptations (cf. timetable and ELS models in §3.3.1 where agents were used to generate large scripts automatically).

- Meta-relations can, in principle, enable further abstraction of algebraic relationships in addition to the structural abstraction using prototypes (and template

definitions). Meta-relations and prototyping can be used to achieve similar abstractions but with different characteristics (cf. square examples in listings 4.20 and 4.27). Recursion is easy with meta-relations but would require a lazy copy mechanism if done using the prototype approach, which is how Subtext supports recursion [Edwards, 2005]. The lazy copy of Subtext is actually a form of meta-relation in any case, but one that perhaps relates better to the concrete nature of construals than the notion of meta-relation which is not observable. Further work is needed on both of these approaches (cf. chap 8), however neither is possible with EDEN and one or both would prove vital for supporting more abstract programs.

- Agents could, in the future (cf. §8.2.5), take advantage of the semi-structured nature of the OD-net using schemas and capability-based security to provide fine-grained restrictions on observation and interaction. Such restrictions embed an intended protocol of interaction into the program rather than leaving it open to the arbitrary subjective change of a construal (cf. problem B6 in §3.3.1). Restrictions of this kind provide assurance and help to guarantee objectivity. LSD is the current EM approach for specifying restrictions on agent interaction but is problematic (cf. §3.2).

It is not difficult or problematic to consider the OD-net described in this chapter as a way of representing programs. Indeed many who have only been briefly exposed to Cadence and the DASM notation have mistaken it for a traditional programming language and environment. The conclusion here is that Cadence does support "programs" and that it is a substantial improvement over EDEN, even though much work is required on tool development.

6.3.3 Supporting Migration from Personal to Public

Having argued how Cadence supports both construals and programs it is comparatively trivial to show how to move from one to the other. The development process outlined

in §6.2 already shows how through experimental evolution a construal can solidify into a program. Over time the modeller can start identifying patterns such as prototypical objects or meta-relationships which occur often. Gradually agent interactions with models become ritualised and through the use of capabilities and schemas this can be captured.

Cadence is a gentle-slope system (cf. §2.1.1). There is no sudden leap from construal to program, different parts of the same artefact may evolve at different rates but still work together. Prototypes can be made and used as they are identified, but concrete singletons[34] can also remain. Relationships can be enumerated and then later converted to meta-relations or prototypes if desired, all without radical translation steps, remaining live and interactive throughout (cf. liveness in §2.1.1). The use of the single underlying OD-net and agent framework for both construals and programs means that the same skills are used for developing both. The development of a program (or construal) is largely similar to how the program is used, through interaction and observation of observables and relationships, the only difference being the sophistication of the changes being made.

The development process described in §6.2 and the Stargate and Kinesin models of chapter 5 all demonstrate a migration from construal to program. Although essentially the same as the EM process, the earlier discussion and the models illustrate how the migration supported by Cadence can go much further than is possible with EDEN.

[34] Singletons are unique objects which do not need to be abstracted into a prototype since no other instance of them is required.

Chapter 7

Cadence and Empirical Modelling

With the Cadence framework being identified in chapter 6 and a prototype with examples given in chapters 4 and 5, it is now possible to analyse Cadence through comparison and independent use. In doing this two additional prototype systems were developed to compare Cadence with EDEN and EM. The first is a new custom notation for EDEN that attempts to implement the Cadence semi-structured OD-net in EDEN. The second is a hybrid tool of EDEN and Cadence to see how they complement each other. Using these tools and the Cadence of chapter 4, students have developed various models for coursework which can be used to evaluate the Cadence concepts. Whereas the models in chapter 5 were developed in conjunction with the development of Cadence for testing purposes, the models in this chapter have been developed post development of Cadence specifically to compare and provisionally evaluate. Additionally, myself and Beynon have also developed smaller models to help compare and contrast EDEN and Cadence. This chapter outlines the two new Cadence systems, how Cadence has been taught to students and how they have used and understood Cadence for EM. The intention being to illustrate and justify the framework given in chapter 6.

7.1 Cadence-in-Eden

Previous developers of the EDEN tool added the ability to define custom definitive notations using an Agent-Oriented-Parser (AOP) [Harfield, 2006a, 2003]. Since definitive notations are the intended way to extend EDEN it seems appropriate to attempt to add the Cadence framework, or some subset of it, to EDEN by making use of a custom definitive notation. The purpose of doing this is to help bridge between existing EM tools and Cadence for the benefit of those familiar with EDEN and for the benefit of existing EM models in EDEN. It also enables a better critique to be given, asking why a new notation is not good enough. The notation is called Cadence-In-Eden (CINE). Key aims for the notation are:

1. Provide an object like structure for organising observables.
2. Enable context switching.
3. Allow for the cloning of objects to generate larger models from prototypes.
4. Reduce type restrictions enforced by the Eden notation itself.
5. See how well the notation can link with other notations.
6. Check out the performance and complexity of such an approach.

All of the above will be used to evaluate the approach. Whilst dynamical definitions could be implemented in Eden by having a new observable for each instant in time, this would in practice lead to an exceptionally large number of observables being generated. As a result implementing dynamical definitions in CINE is unrealistic and indicates that adding a new notation is not the ultimate solution. However, seeing how EDEN copes with object concepts is important.

CINE is similar to the DASM notation described in chapter 4 so it will not be covered in depth here[1]. Internally the parser attempts to generate object-like structures

[1]The most significant differences being the use of *dot* instead of a space between labels and the requirement of a semi-colon at the end of each line. Both are required by the AOP.

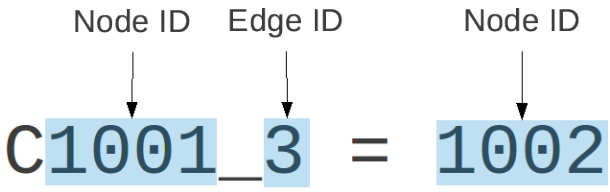

Figure 7.1: An example of an automatically generated Eden observable name using the various ID components. Note that a node id and edge id are actually the same kind of id.

on top of the flat Eden name-space by generating observable names which have two components, a node number and edge number as shown in figure 7.1. These numbers can also be stored in these observables as values and hence it corresponds to the ϕ of eq. 6.7. There is then a mapping from these node numbers to strings and integers using a lookup table (cf. figure 7.2), needed to transcend the limited EDEN type system. An example of the CINE notation is given in listing 7.1 along with its subsequent translation into Eden code in listing 7.2. Figure 7.2 shows the translation mechanism as well as the *names* table used to convert to id numbers.

```
root.table = <1001>;
root.table.sides = <1002>;
root.table.width = 300;
root.table.height = 250;
```

Listing 7.1: Cadence Notation Example.

```
c1_2 = 1001
c1001_3 = 1002
c1001_4 = 5
c1001_6 = 7
```

Listing 7.2: Translation of listing 7.1 into Eden.

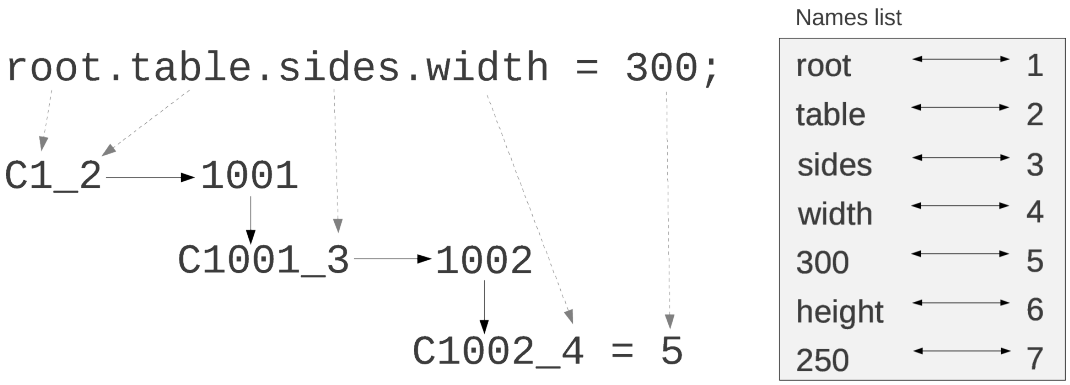

Figure 7.2: An example of the CINE to Eden translation process which involves navigating an existing structure. The names table is also given which corresponds to the examples given.

The short example given in listing 7.1 is an extract from a model of a room adapted from a previous model that used the Donald notation [Yung, 1989]. The room model contains clearly identifiable objects such as a table, light, door and the room itself. Donald provides some concept of structure, much like Scout (cf. §3.3.3), but this is not well mapped into Eden and is certainly not generic enough for use across other notations. As a consequence, the objects in the original model cannot be clearly identified from within Eden. One of the major drawbacks of EDEN is that all communication and connection between different notations must go back to the underlying Eden translation of that notation (cf. issue E1 in §3.3.3 and §6.3.1). The CINE notation is also translated into Eden, so to reduce the impact of this on inter-notation communication, several utility functions have been provided in Eden for the CINE notation as otherwise the modeller would need to understand the automatically generated observable names given in listing 7.2. The use of a selection of these functions is illustrated in table 7.1.

Function 4 in table 7.1 enables the result of a path to be queried, with the path being given as a list of strings where the strings label nodes and edges. The first element of the list is the starting node and the rest label edges to follow from that. If numbers are given instead of strings then they are interpreted directly as node ids. The last example

Table 7.1: CINE notation and the equivalent Eden expressions

root.a = 5	`observable("root","a") = convertname("5");
root.c.d = 5	define(query(["root","c"]),"d","5");
root.b **is** root.table.height	define("root","b",["root","table","height"]);
?root.table.width	query(["root","table","width"]);
... **is** root.table.height	b **is** navigate("b", ["root","table","height"]);

in table 7.1, *navigate*, is the same as *query* except that it will also add dependencies to an observable so that if any part of the path were to change that observable would be marked as out-of-date. In the example the observable *b* is being defined and so the first parameter to the navigate function is the string form of that observable name so as to identify the observable to add the dependencies to. Such an approach seems convoluted but this is due to EDEN not adding dependencies on observables that are used inside functions (cf. issue D1 in §3.3.2) and so a manual approach is required that makes use of a previously little used feature of Eden[2].

Within the CINE notation itself it is possible to write definitions that are similar to those in the real Cadence tool[3]. Internally definitions translate to the navigate example given in table 7.1. Listing 7.3 is an example of a definition written in CINE with its Eden translation shown in listing 7.4.

```
root.a is root.table.width;
```

Listing 7.3: Cadence notation definition example.

More complex nested paths may be used in definitions by using parentheses in CINE and nested list structures in the Eden form. One good example of this is for

[2]The feature used is $\sim>$ which allows a function to explicitly add dependencies. Unfortunately it was found that this feature did not work correctly and so the EDEN source needed to be patched before the CINE notation could be used (tkeden-1.72).

[3]Not including dynamical definitions.

```
c1_8 is navigate("c1_8", ["root","table","width"]);
```

Listing 7.4: Translation of listing 7.3 into Eden.

conditional statements where the condition is used to select an edge in another object. Listing 7.5 shows an *if*-statement in the CINE notation[4].

```
root.a = true
root.b = 2
root.c = 3

<1000>.true is root.b
<1000>.false is root.c
root.d is <1000>.(root.a)
```

Listing 7.5: An if-statement in CINE showing nested queries.

```
c1_10 is navigate("c1_10",[1001,["root","a"]]);
```

Listing 7.6: Translation of listing 7.5 into Eden.

It is clear that to a large extent the power of object structures and the navigation style definitions in the Cadence framework can be added on top of the EDEN environment as a new notation, but the question remains of how useful it is in this role. If any structures described in CINE are then flattened into unintelligible observables in Eden it seems that all benefits are lost[5]. One final capability of this notation is that of being able to clone an object and therefore automatically generate large numbers of Eden observables. Unfortunately the observables being generated are of the kind in figure 7.1 so is unhelpful to other notations (or the modeller). To take advantage of CINE (cloning

[4]Note how in CINE there is no syntactic sugar for if-statements as there is in DASM.

[5]It becomes nearly impossible for the modeller to interpret the observables as structure without assistance and also requires the modeller to understand the translation mechanism in figure 7.2.

etc) the other notations would need to translate to it and work directly with it, and in doing this the Eden language would be made redundant as the meta-domain language, CINE would take its place.

In order to test the real capabilities and limitations of the new notation before moving on to the next approach (cf. §7.2), a few different types of model were constructed, some of which will be outlined in the following subsections. Each of these models is able to illustrate the benefits of having the Cadence framework applied to EDEN, however they also show that CINE *as a notation* cannot be fully exploited.

7.1.1 Cloning for Timetable and Bubble Sort

One such model developed was an attempt to fix and improve an existing timetabling model (cf. figure 7.12). In this model there are many almost identical display elements for each slot and originally these needed to be manually copied. The author of the original model had attempted to make a form of object cloning to achieve this, however, it has become too complex to understand and adapt. To try out the new CINE notation an attempt was made at performing this cloning activity using CINE.

Due to the complexity of the timetable model a simpler model of bubble sort was constructed first which included similar box like visual elements as the timetable. The first step was to develop a representation of the display elements inside the CINE notation. A prototype point, line and box was developed, as shown in listing 7.7. The tilde operator in CINE means make a clone of the object on the right-hand-side[6]. The last line of listing 7.7 shows one of many definitions relating the individual corner points of the box to the boxes overall position using simple (built-in) arithmetic operations.

With the prototype in place all of the individual boxes to be displayed, representing the cells in an array, can now be cloned. The first cell is cloned from the prototype box (cf. listing 7.8) but all subsequent cells are cloned from cell 1 or 2 to further simplify the script (cf. listing 7.9).

[6]Cloning *null* creates a new empty object.

```
...
proto.line ~ null
proto.line.p1 ~ proto.point
proto.line.p2 ~ proto.point
proto.line.width = 1
proto.line.colour ~ proto.colour
proto.line.type = line

proto.box ~ null
proto.box.type = shape
proto.box.N ~ proto.line
proto.box.S ~ proto.line
proto.box.E ~ proto.line
proto.box.W ~ proto.line
proto.box.x = 0
proto.box.y = 0
proto.box.width = 0
proto.box.height = 0
proto.box.linewidth = 0
proto.box.colour ~ proto.colour
proto.box.N.p1.x is this.parent.parent.x.sub.
    (this.parent.parent.width.div.2)
...
```

Listing 7.7: Line and box prototypes in CINE.

```
bubble.array.1 ~ proto.box
bubble.array.1.width is bubble.array.bwidth
bubble.array.1.height is bubble.array.bheight
bubble.array.1.x = 100
bubble.array.1.y = 100
bubble.array.1.linewidth = 1
```

Listing 7.8: First cell in bubble sort array.

```
bubble.array.2 ~ bubble.array.1
bubble.array.2.prev = bubble.array.1
bubble.array.2.x is this.prev.x.add.(this.prev.width).add.
    (bubble.array.space)

bubble.array.3 ~ bubble.array.2
bubble.array.3.prev = bubble.array.2
```

Listing 7.9: Second and third cells as clones.

Figure 7.3: Bubble cells from CINE script. The last cell on the right is red.

These structures now exist within the CINE notation and have been translated into the flat Eden name-space[7]. To visualise these boxes it is necessary to utilise the existing Donald notation. There are two ways to achieve this: 1) link the CINE observables by dependency into a Donald script, or 2) automatically generate (via an agent) a Donald script from the CINE structure. The first approach involves entirely replicating the CINE structure in Donald form and then connecting with dependency[8]. This completely removes all benefit of cloning gained and involves the storage of duplicate representations of the same thing. The second approach is far less interactive since any change in CINE will not be immediately visualised until a new script is generated and executed. Neither approach is desirable. However, automatic script generation has been attempted and results in the bubble array cells being drawn as in figure 7.3.

To change the colour of a cell (as with the last cell of figure 7.3) the change in listing 7.10 must first be done in CINE and then a new Donald script generated using the command in listing 7.11 (an Eden procedure).

[7]The actual representation is always in EDEN, CINE is only a view of EDEN.
[8]An example of inter-notation communication difficulties and notation inconsistencies (cf. §3.3.3).

```
bubble.array.8.colour.R = 255;
```

Listing 7.10: Changing a cells colour in CINE.

```
makescript(_cadence_query(["bubble","array"]));
```

Listing 7.11: Updating the Donald display.

Due to the infeasibility of this script generation approach and the complexity of the Eden translations for CINE, the use of CINE for the timetable model was abandoned. The cloning mechanism and the way of structuring the model is faithful to Cadence and works well. However, it is the inter-notation communication problem that prevents it working in practice. Having to generate scripts goes against the liveness principle (cf. §2.1.1) which is fundamentally important for Empirical Modelling. Despite attempts at getting the script to be generated automatically upon changes to CINE, it was deemed to be too slow and inefficient for practical use.

7.1.2 Boolean Lattice Model

The CINE notation may be used for certain kinds of model largely independent of other notations and hence the associated translation problems. A model of a boolean lattice is well suited to being represented in the graph-like way of Cadence. Figure 7.4 shows a visualisation of a boolean lattice that has been described in CINE[9], with the right-hand image being a particular sublattice extracted as a subgraph in CINE.

The lattice can now be manipulated and observed in different ways either by changing the structure in CINE or through procedural actions using the Eden functions provided for interaction with CINE[10] (cf. table 7.1). These kinds of models are difficult

[9] There is some connection to Donald for this visualisation, making use of the script generation technique described above

[10] The need to use special functions in Eden for manipulating CINE structures is an example of issue E4 in §3.3.3

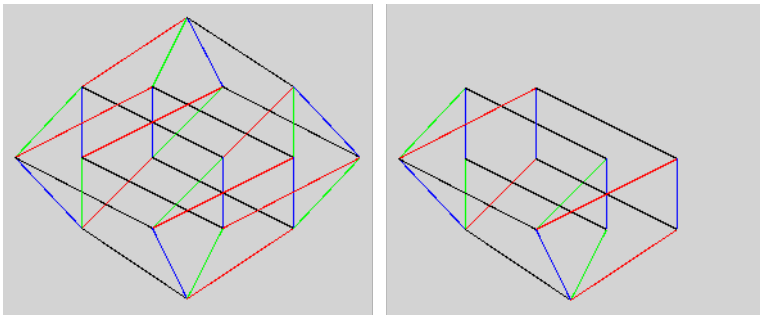

Figure 7.4: Boolean lattice described in CINE. Left is full lattice, right is a subset of the lattice.

when just using Eden, however, a notation called ARCA does already exist[11] for graphs of this kind.

An important feature of the CINE construal of the Boolean lattice is that it allows the modeller to refer to the lattice, and to manipulate it, in ways that reflect the perspective of a mathematician. Specifically, it is easy to reconfigure the structural definition using the Cadence-in-EDEN notation in order to describe constructions that have mathematical interest. These include:

- sublattices, such as may be specified by selecting a subset of the generators;
- quotient lattices, such as may be specified by identifying certain subsets of generators;
- decreasing subsets of the lattice, comprising all those elements that are less than or equal than at least one of a set of non-comparable elements of the lattice.

Decreasing subsets of the Boolean lattice depicted in figure 7.4 feature in the construction of the free distributive lattice on 4 generators, as illustrated in the EDEN model [Beynon, 2003]. The use of CINE leads to a much simpler and more elegant construction than was developed in the EDEN model.

[11] Although not available in newer versions of EDEN.

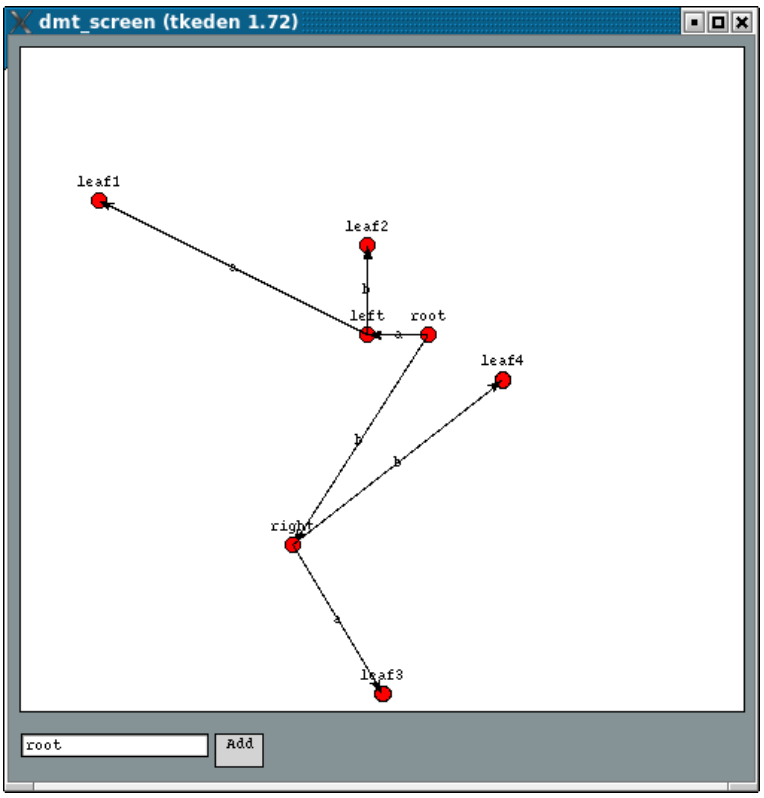

Figure 7.5: DMT visualisation of a CINE structure.

7.1.3 Visualisation using the DMT

Taking advantage of, and adapting, an existing EDEN model (the DMT [Harfield, 2006b]), a graph visualisation of any CINE structure can be generated. Originally the DMT was used to visualise EDEN dependency relationships as a graph but with only minor alterations it can traverse the CINE observables in EDEN to show structural relationships. Figure 7.5 shows the DMT visualisation of the CINE structure given in listing 7.12.

Visualisation of this kind is not included in the Cadence prototype of chapter 4 but shows how Cadence graphs can be automatically visualised. More significantly here though, it is an illustration of custom mediator agents (EDEN procedures) observing ϕ and developing one possible, highly generic, representation of it.

```
root.a = left
root.b = right
right.a = leaf3
right.b = leaf4
left.a = leaf1
left.b = leaf2
```

Listing 7.12: Test binary tree in CINE (cf. figure 7.5).

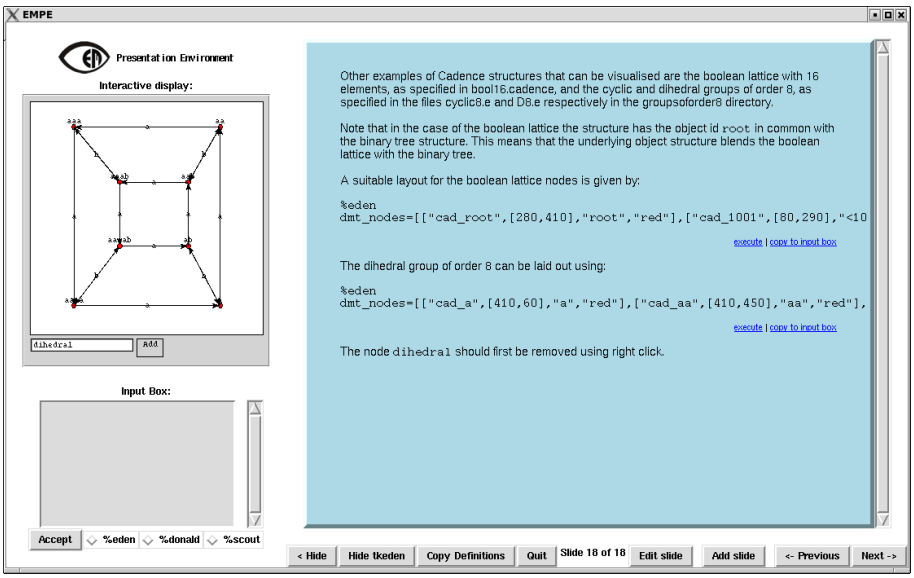

Figure 7.6: Dihedral group of order 8 in CINE, visualised using DMT and embeded into a presentation.

Taking advantage of the DMT visualisation of CINE structure, a model of the dihedral group of order 8 was created and then visualised (cf. figure 7.6). Unlike the boolean lattice model in §7.1.2 where a custom visualisation was developed, here the generic DMT was used. This illustrates how the modelling of structure is a generically useful capability in its own right, rather than only being useful for particular visualisations[12].

It is interesting to compare the use of CINE in the construal of the dihedral group of order 8 with EDEN construals of group structures that have been developed using the ARCA notation (cf. [Beynon, 2003] and the EDEN model of Schubert's Erlkoenig [Beynon, 2006a] discussed in [Beynon, 2006b; Beynon et al., 2006b]). ARCA [Beynon, 1983, 1986a] was the first definitive notation to be conceived, and is unusual in that it incorporates ways of manipulating references to group elements as values. By moving away from the conventional association of value with identifier such as is typical of traditional programming languages and definitive scripts, Cadence is able not only to emulate this feature of ARCA, but to support much richer and more general modes of reference and manipulation.

So structure is useful for certain kinds of model and CINE works well when there is minimal inter-notation communication. However, for cloning, visualisation and general model management it is not practicable to have it only as a notation on top of EDEN. Putting the Cadence framework above EDEN fails to resolve the key issues identified in §3.3[13] and, therefore, the Cadence framework must form the foundations of an EM tool in order to benefit fully from it. The ϕ (cf. eq. 6.7) function needs to be considered, it is argued here, as the most primitive mode of representation.

[12] Previously structure was used in Donald and Scout but was not often utilised as a core part of a model.

[13] Some can be resolved but at the expense of exacerbating others.

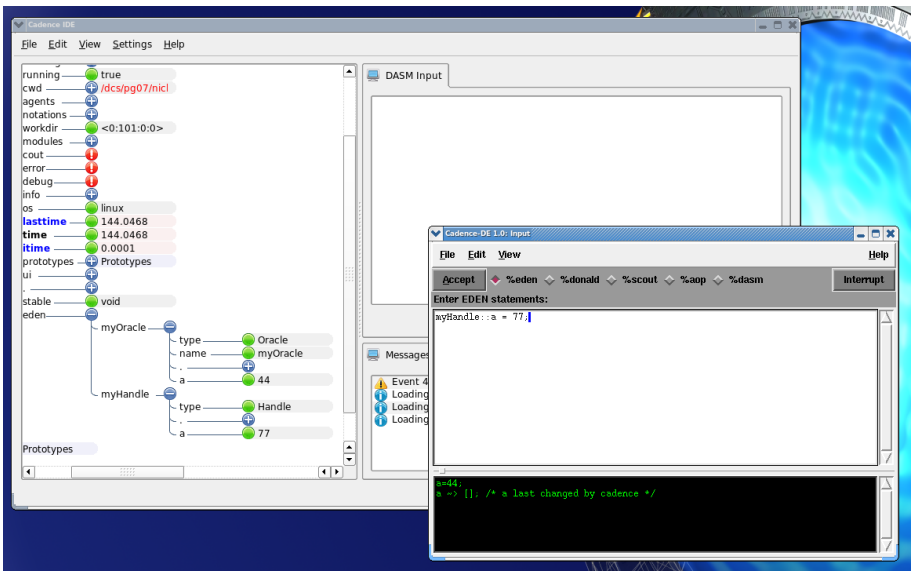

Figure 7.7: Screenshot showing EDEN running inside Cadence and communication between the two tools.

7.2 Eden-with-Cadence

An alternative approach to constructing a new notation within EDEN is to develop a hybrid where both the Eden and Cadence tools are combined and given mechanisms for inter-tool communication. The full benefit of both environments is then available and given sufficiently well developed communication mechanisms the powers of each can be utilised together to produce new kinds of model that might previously have been too complex or impossible. The issues of translation that exist with CINE are no longer relevant and the modeller is free to customise the communication. Whilst Cadence should be capable of all that EDEN is capable of in principle, in practice it is a far less well developed prototype and lacks the extensive library of features and past projects that EDEN has. The benefits of CINE also apply here, Cadence can be used to enhance existing models in EDEN.

The hybrid was achieved using the Cadence module mechanism. EDEN was converted into a Cadence module that could then be loaded dynamically into an ac-

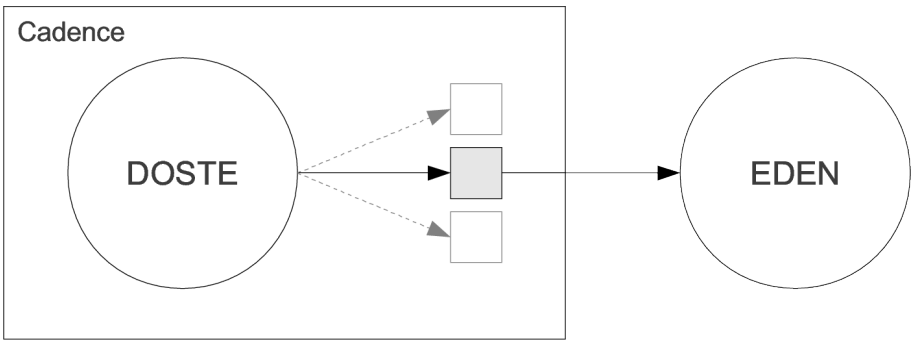

Figure 7.8: Architecture of Eden-with-Cadence hybrid.

tive Cadence environment as desired. This involved minor modifications to the main run-loop of EDEN so that it became driven by Cadence rather than being its own application. Additional changes and simplifications were required to support inter-tool communication[14]. The diagram in figure 7.8 illustrates this architecture.

7.2.1 Inter-Tool Communication

The module (shaded grey in figure 7.8) is a C++ file implementing a communication mechanism between the two tools, and is approximately 200 lines. Two mechanisms were developed to enable the two tools to communicate with each other. Both of the approaches involved the sharing of observables but did this in different ways:

1. Mapping the EDEN symbol table directly into the Cadence graph (cf. figure 7.9).
2. Using an LSD style agent mechanism based upon oracles and handles (cf. figure 7.10).

The first approach would have enabled an almost seamless means of accessing all EDEN observables from within the Cadence graph. A Cadence handler was developed that routed events from Cadence destined for EDEN observables into the EDEN tool where they retrieved or modified the relevant observables. Ultimately this approach failed

[14]The existing virtual agent mechanism was adapted to provide name-spaces that could be used as contexts and mapped into Cadence.

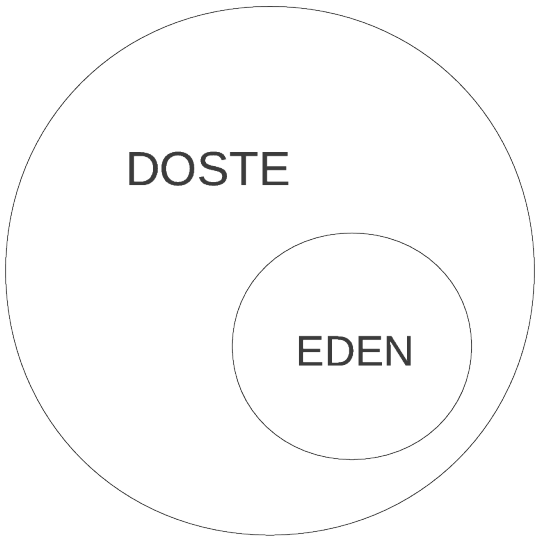

Figure 7.9: The EDEN symbol table is mapped into the DOSTE graph to merge the two tools at the lowest level.

due to the disparity between type systems and the definition evaluation mechanisms. Since an EDEN observable could be given both an EDEN definition and a Cadence definition there was potential for serious conflicts to develop. Also, EDEN observables could not contain objects and so broke when such things were attempted from within Cadence. This low-level merging was eventually abandoned in favour of the second approach.

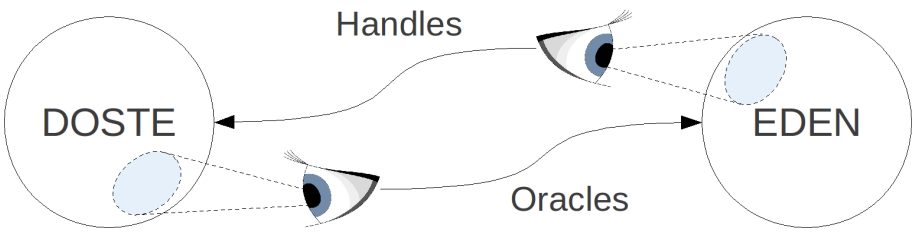

Figure 7.10: Oracle and Handle approach to inter-tool communication. Handles observe EDEN and change DOSTE whilst Oracles observe DOSTE and change EDEN.

With the second approach agents were created in Cadence which monitor certain nodes in the Cadence graph and certain name-spaces in Eden (a feature enhanced as a part of this work). When changes are noticed by these agents they then perform a corresponding change in the other tool. They act as intermediaries that provide Eden with oracles and handles to the Cadence environment. An oracle is something an agent can observe, whilst a handle is something an agent can change[15]. Whilst some typing issues still remain they are easier to resolve using this more relaxed approach to communication.

The *oracle* and *handle* agents are specified inside Cadence as shown in the examples of listings 7.13 and 7.14. Both types of agent must be placed inside an *eden* context so that the EDEN module can create the appropriate agents.

```
.eden myHandle = (new
    type = Handle
);
```

Listing 7.13: EDEN handle for Cadence specified in DASM.

```
.eden myOracle = (new
    type = Oracle
    a = 1
);
```

Listing 7.14: EDEN oracle for Cadence specified in DASM.

Inside Eden it is now possible to create and change observables inside a myHandle namespace as well as observe changes to observables in a myOracle namespace. Listings 7.15 and 7.16 give Eden statements corresponding to the handles and oracles defined in listings 7.13 and 7.14.

[15]Both oracle and handle are terms borrowed from LSD.

```
myHandle::a = 2;
```

Listing 7.15: Eden script changing a Cadence handle.

```
?myOracle::a;
```

Listing 7.16: Eden script observing a Cadence oracle.

By separating observables into oracles and handles rather than mixing the two it creates read-only and write-only observables. As a consequence the problem of multiple definitions on a single observable disappears since changing a read-only oracle in Eden by giving it a definition will have no effect upon Cadence and such a definition will be removed by Cadence when it changes that oracle. The opposite it also true for handles.

The following sections discuss a few of the models developed using the hybrid tool and communication mechanism discussed here. Table 7.2 lists features that are supported by EDEN and Cadence. From the table it is clear that neither tool is comprehensive (at present) and so the models discussed illustrate ways in which each can be taken advantage of.

7.2.2 Traffic Light System

Originally this model was developed by James McHugh, an MEng student, as a part of his coursework for the Empirical Modelling module at Warwick. It used an older version of the Cadence-EDEN hybrid that still utilised the first inter-tool communication approach of merging symbol tables. More recently McHugh's model has been updated to make use of the newer oracle-handle approach to connecting the tools together. The model itself is of a T-junction with traffic lights and shows cars and queues to see the effect of altering the traffic light sequence. McHugh's objectives were to improve visualisation, realism and increase the complexity of related older traffic models. For this reason

[16] Using SASAMI notation but is far more basic than the game library of Cadence

Feature	Cadence	EDEN
Structural relations	✓	X
Latent relations	✓	✓
Dynamical relations	✓	X
Custom Agency (live)	X	✓
Custom Functions (live)	-	✓
2D line drawing	X	✓
3D models	✓	✓[16]

Table 7.2: Comparison between EDEN and Cadence

Cadence was used for visualisation and animation using dynamical relations.

It is possible to construct such a traffic model entirely in Cadence, however, McHugh gives a reasonable argument for making use of the hybrid. His argument is that a real traffic light system would likely be controlled by a procedural program of sorts and so he wished to write that aspect of the model in a procedural way. Custom procedural agents were therefore required and so this feature of EDEN was to be used. For the rest of the model he concluded that Cadence, with its dynamic definitions and objects (with cloning), would be more suitable for representing cars and their motion. For the purposes of this section the use of the hybrid features will be focused upon.

Listing 7.17 gives a script extract from the newly updated McHugh model[17] which links an observable in Cadence with an observable in EDEN by dependency.

In the model there is procedural Eden code linked to a timer that calculates the light sequence (cf. listing 7.18). There are 3 observables, state1, state2 and state3 which correspond to each of the traffic lights at a 3 way junction. Each light can be in one of 4 states which correspond to which lights should be active (the observables can have the values 0,1,2 and 3). In Cadence handles and oracles have been set up in an 'eden' context and so a definition has been used to connect the EDEN state observable to

[17]Update done by myself in December 2010.

Figure 7.11: James McHugh traffic light model

```
@display %deep light1 = (new union (@tlight)
    x = 339
    y = 330
    stage is { @root eden handles state1 }
);
```

Listing 7.17: DASM script connecting Cadence and EDEN in the Traffic model.

the corresponding state observable for that light in Cadence (cf. listing 7.17). Other observables have been used in Eden as oracles which allows Cadence to control aspects of the Eden code.

What this shows is the successful integration of a procedural script written in Eden with the Cadence OD-net. Agents can control and observe specific observables. It is a seamless integration of two paradigms and two tools using dependency. It also shows that having just a single paradigm, as Cadence does at the time of writing, is not always desired and that allowing for a mixture of paradigms is the way forward. In this respect it highlights some limitations of the current Cadence implementation (cf. chapter 8). At the same time it also justifies the existence of dynamic definitions and object structures as well as better integration with more sophisticated graphics libraries and so on. Without these newer capabilities such a model would be a challenge to construct in this more realistic way.

7.2.3 Timetable Revisited

In the late 90's a group of EM researchers, led by Beynon, started a project that involved developing a timetabling model in EDEN which the department secretaries could then use to organise project orals [Keen, 2000; Beynon et al., 2000b]. As can be seen in figure 7.12, this model involved a large number of almost identical cells and other components. Ordinarily each one of these cells would need to be modelled, most likely involving a lot of copy paste operations in a script. Due to the scale Allan Wong decided against doing this as it was inflexible to change. Instead he developed a mechanism which allowed him to effectively clone some existing default observables and automatically construct all the required observables by iterating a procedure. The implementation of this has since proven to be very complex and difficult to understand and is not generic in nature.

The model is an obvious candidate for trying out the object and cloning characteristics of Cadence, and following the failed attempt at using CINE it seemed appropriate to try again with the hybrid. Beynon and myself have partially developed a new version

```
proc Tlight1:iclock {
   if (state1==4){
      if (state2==4 && state3==4 && turn==0){
         state1=3;
         p1=4;
      }
   } else if (state1 == 3) {
    oticks1++;
    if (oticks1 > otime1) {
        if (p1 == 2) {
            state1 = 4;
            turn=1;
        } else {
            state1 = 2;
        }
        oticks1 = 0;
    }
   } else{
      gticks1++;
      if (gticks1>gtime1){
         state1=3;
         p1 = 2;
         gticks1 = 0;
      }
   }
}
```

Listing 7.18: EDEN agent controlling a traffic light. *state1* is a handle observable for Cadence (cf. listing 7.17).

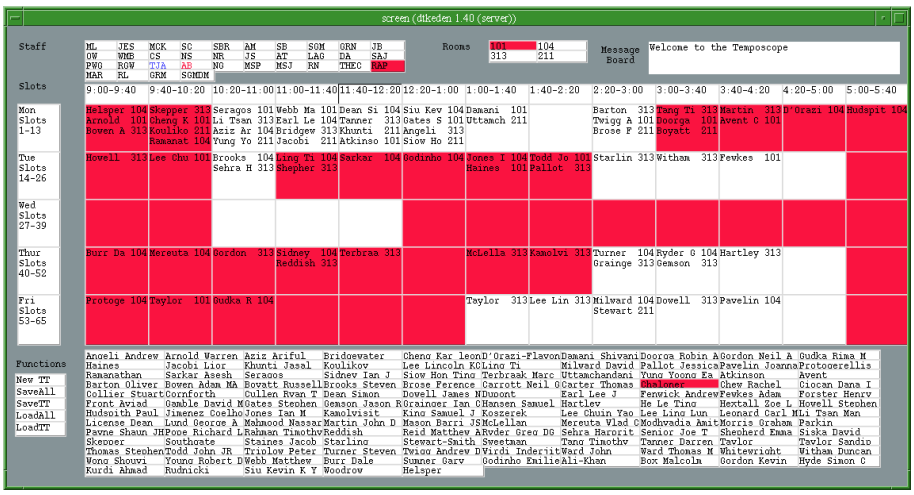

Figure 7.12: Original Eden timetable model (Chris Keen version)

of the timetable model where the cells are represented in Cadence but the visualisation and interface still make use of Donald in EDEN. The intention here was to be able to clone cells and have an automatic mechanism by which these cloned Cadence objects get converted into the required Donald observables in Eden.

```
@slotswin slotnum is {@root eden output sn};

.eden input = (new
    type = Oracle
    x is {@timetable slotsTable (@day (@slotswin daynum))
                                (@slotswin slotnum) x}
    ...
);
```

Listing 7.19: DASM context selection for cells.

To generate the Donald visualisation, context switching was used in Cadence to cycle through all the cell objects (cf. listing 7.19). The currently selected context (controlled by Eden, as shown in listing 7.20), or cell, has its various attributes connected

```
winix = 0;
sn is winix % 13;
output::sn is sn;
```

Listing 7.20: EDEN selects the context with a handle observable.

```
x is input::x;
wloc is "\tbox␣:␣" // "[{" // str(x) // "," ...;
```

Listing 7.21: Generating Donald script from Cadence oracles.

to an Eden Oracle agent so that when a context changes Eden will be able to observe that new cell. In Eden, various observables with dependencies on these Oracles build up a Donald script using string operations and finally a triggered procedure is used to execute this generated Donald script (cf. listing 7.21). The result is a considerably more elegant and generic way of generating large numbers of similar display objects.

Whilst this approach to generating Donald scripts from Cadence structures is interesting, it suffers from several problems that, as with CINE, mean that the approach is less than satisfactory:

1. Oracle observables need to be manually constructed, although this could in principle be automated by some other means.

2. Inter-tool communication is not responsive enough. There were problems with changing observables too quickly and for certain cells to be missed. Other models have also demonstrated this problem[18].

3. No direct dependency links. If something changed in Cadence then EDEN would not be aware of it until a new Donald script was manually regenerated.

4. Need to flatten the Cadence structures and rebuild object-like structures of Donald.

[18] A hybrid (Cadence-EDEN) traffic model by James Brotherton-Radcliffe skipped updates seemingly randomly, causing the logic to break.

The conclusion that can be drawn from this is that using EDEN for visualising Cadence suffers from the same problems as CINE. Either everything needs to be fundamentally based upon Cadence to take advantage of its structural characteristics, or a far more sophisticated hybrid is required. However, the use of EDEN agency to control and manipulate Cadence has potential for simpler models.

7.3 Cadence in the EM MSc Module

Having seen how both the CINE notation and the hybrid fail to resolve the issues of §3.3, and that EDEN cannot be modified to support the Cadence framework of chapter 6, it is now time to revisit the Cadence prototype of chapter 4 as a standalone environment. The question is: *Can Cadence be used for Empirical Modelling?* To answer the question, this section explores how Cadence has been used in the teaching of Empirical Modelling.

Both the 2009 and 2010 class of MEng and MSc students doing the CS405 Empirical Modelling module have been introduced to Cadence and the Cadence-EDEN hybrid tools. Each year has seen approximately 40 students so a total of 80 students have used Cadence and the hybrid, along with other 3rd year project students. For the module the students had roughly 8 lab sessions of 2 hours each and around half of these were devoted to Cadence or the hybrid tool. Also, several of the lectures discussed Cadence and EM tools generally to explain the conceptual aspects of them. The nature of the labs and coursework will be given here along with some feedback and example models from the students.

7.3.1 Teaching Cadence

The lectures for the module primarily focused on the conceptual aspects of Empirical Modelling in general. However, a few lectures took a look at Cadence in an attempt to explain how it relates to the EM conceptual framework and how it should be used[19].

[19] These lectures were given prior to the full development of the Cadence framework given in chapter 6.

The following Cadence related topics were covered by the lectures:

- Multiple paradigms, including: prototype-based objects, data-flow, reactive and functional.
- Graph terminology and computation as navigation concept. Material for this was taken from chapter 4 of this thesis.
- Merging of requirements, design, implementation and testing into a single environment. Not entirely specific to Cadence but was used to explain the intention of Cadence.
- Dealing with different contexts. Switching between models or parts of models better supported in Cadence due to the object mechanisms.
- Dynamic definitions. The benefits and difficulties of having process observables.
- Characteristics of the Stargate model. The different agent views and the way state has been modelled.

7.3.2 Lab Sheets

For 2010 there were 5 labs based around the Cadence tool, as outlined below [Pope and Beynon, 2010a]. The labs are not assessed but are used to teach the students how to use tools they will later require for their coursework. It was decided by Beynon, the module organiser, that Cadence provided a more up-to-date and exciting tool for students to use despite its prototypical nature. One of the purposes for using Cadence was to see how well the students took to it and what kinds of models they would eventually be able to construct using it. In addition, whilst the students were working with the tool it allowed myself to develop better interfaces for them to work with and find bugs that had not previously been seen.

1. **Introduction to the EM tools** The first lab of 2010 introduced the Cadence tool using the Stargate model discussed in chapter 5. Students were given exercises

that involved observing and modifying the model interactively, becoming familiar with the DASM syntax and the nature of the tool. Some of the semantics of the different definition types was explained.

2. **Lift Exercise** Lab 2 asked the students to construct their own model from scratch in Cadence. The exercises gave examples of how to clone an image object, customise it and put it on the screen. Using these skills they built up a model of a lift which they went on to animate, making it move between floors when buttons were clicked using the dynamical definitions in Cadence. The final exercise was open-ended, asking the student to extend the model in various ways.

6. **Using Cadence and EDEN together** Having introduced EDEN in a previous lab, the Cadence-EDEN hybrid was introduced to the students as a way of supporting agency. The main focus of this lab was on demonstrating inter-tool communication using oracles and handles. They were tasked with constructing a word game model that made use of both Eden and Cadence and involved communication between the two.

7. **Cadence DIY introduction** Although mostly a coursework preparation lab, a question and answer session on Cadence was also given which went over more difficult topics with examples. Questions asked related to the semantics of definitions and how to use the deep cloning feature of DASM.

8. **Networking with Cadence** The final lab of the year asked students to work as pairs on the same model. Similar to the lab on "Using Eden and Cadence together", this lab asked them to use two machines running Cadence to communicate via the networking module. It also made use of EDEN and they were tasked with getting observables of a model in EDEN to connect with Cadence, shared with another Cadence using XNet and finally to have them appear in another instance of EDEN on that remote machine.

7.3.3 Coursework Models

Some of the models developed by students have already been introduced (cf. §7.2.2), however, there are two particular pure Cadence models that are worth mentioning. A SCUBA diving model by William Dangerfield and a Calculator model by David Evans. The calculator model is of particular interest because it demonstrates, to an exceptional degree, the use of Cadence characteristics not found in existing EM tools. This section will explore these two models.

Calculator

The calculator model developed by Evans implements a calculator from binary logic gates (cf. figure 7.13). The logic gates are also modelled in Cadence using the boolean operators as given in listing 4.17 and figure 4.8[20]. Of course, the boolean operators are themselves modelled as structural relations.

What is remarkable about the calculator model is the purity and scale of it. There are over 440k observables[21] involved in the model, with many of those having latent or dynamical dependency definitions, and no use of agency beyond the game library for visualisation[22]. The entire model has been described within the OD-net (Φ). It is an example of a program that has fully transitioned from a construal, although it could be argued that it was program-like from the beginning since there was always a clear objective and little ambiguity or subjective decision making since it is a well understood artefact. However, Evans did not understand all of the components involved prior to development and so it was initially a construal to him. Despite it being program-like from the start, it does show how Cadence can support artefacts that can be considered as programs whilst still remaining flexible. Evans himself states that:

> *"[Cadence] allows any user with some deeper knowledge of the model to directly modify any part of the calculator, for example by causing a bit to*

[20] Evans also extended the existing boolean operators to include *xor*.
[21] The largest EDEN model to date only contains 5k observables so this is almost a 100 fold increase.
[22] Even the LCD screen has been modelled down to the individual segments

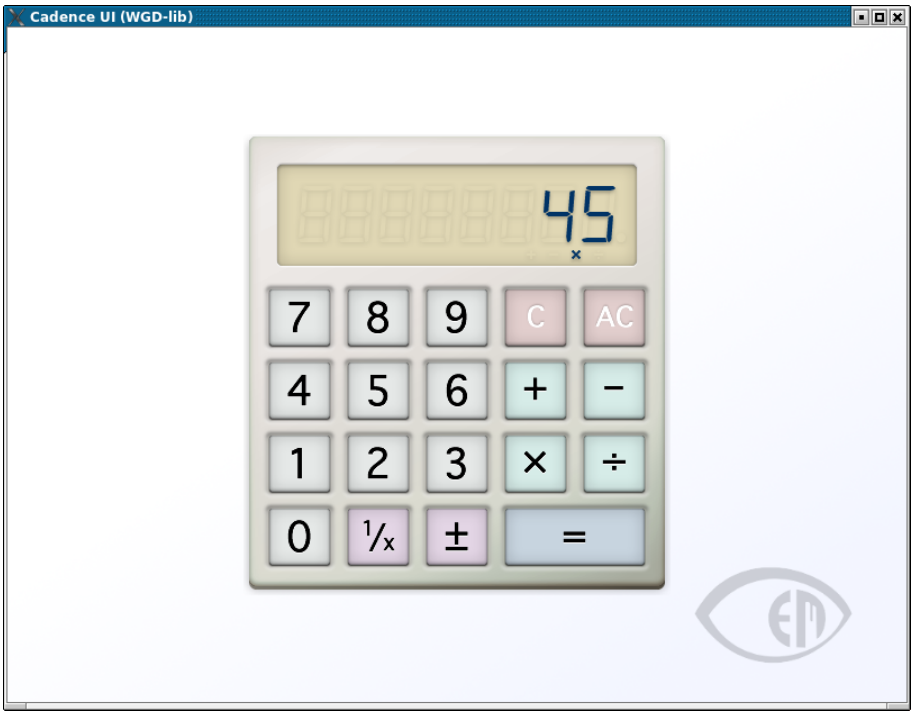

Figure 7.13: David Evans' Calculator Model 2011

be stuck on, or modifying the behaviour of certain buttons to simulate a hardware glitch. The potential ways in which these bit operations might fail can be better understood in this way. Similarly, new functionality can be added..." [Evans, 2011]

The modifications of which he speaks are done live via the Cadence IDE. To manage, comprehend and construct a model containing 440k+ observables requires the use of structural organisation. Evans makes extensive use of Cadence's structural relations and cloning capabilities by, in a hierarchical fashion, constructing individual components such as bit comparison constructs (cf. listing 7.22) which are then combined to produce a byte comparison construct (cf. listing 7.23). The approach taken here is similar to that used by Subtext as described in §2.1.3. Each of these components were tested independently via the IDE and refined from an initially provisional prototype to a functional component. The artefact was decomposed into smaller more manageable models, something that is more difficult to achieve in EDEN (although not impossible). Figure 7.14 shows the prototype hierarchy found in the calculator model.

```
.binaryStuff bit_nequals = (new union(.binaryStuff bit)
    input1 = null
    input2 = null
    b is {.input1 b xor (.input2 b) or (.prev b)}
);
```

Listing 7.22: Bit inequality "function" for calculator model.

Dynamical definitions were also used in the calculator to observe events such as button presses and to store state in register like constructs. This use of dynamical definitions for registers relates strongly to attempts in the EDEN Logic Simulator [Lee, 2007], discussed in §3.3.2, where similar structures could not be constructed without resorting to using agency in an unprincipled way to arbitrarily break cycles.

Evans gives a small critique of Cadence at the end of his coursework paper

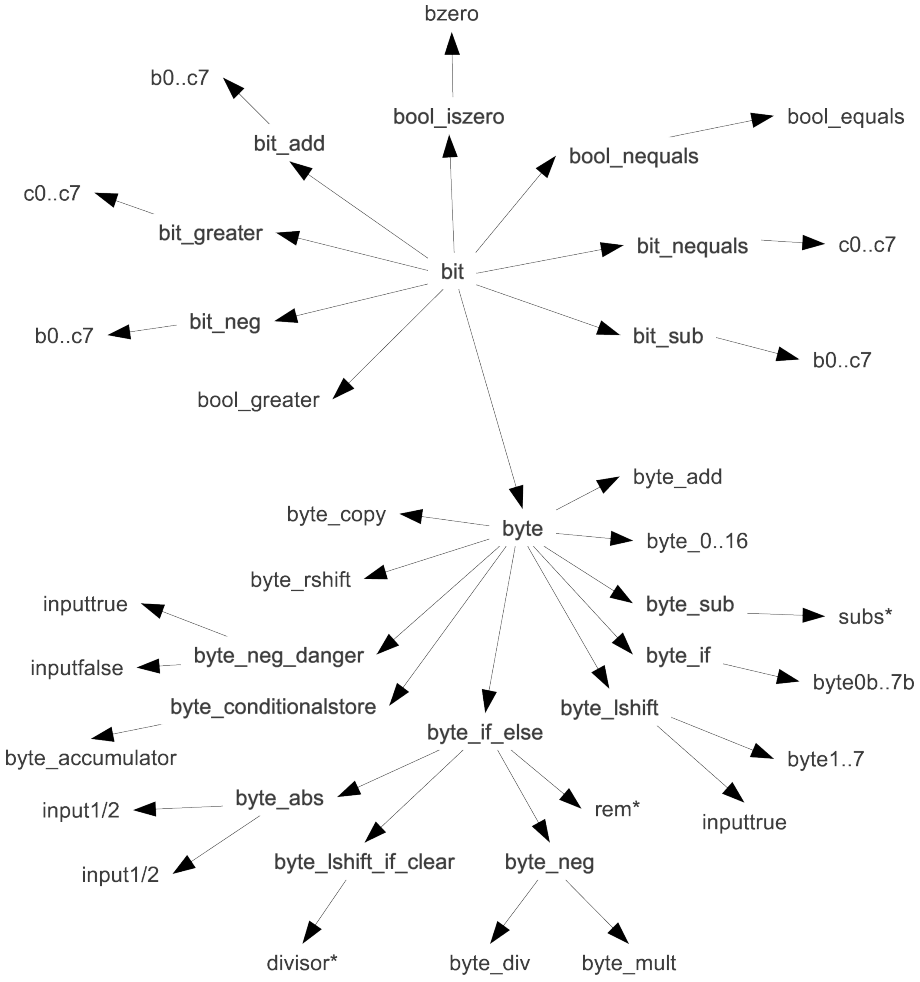

Figure 7.14: Partial prototype hierarchy for the Calculator model.

```
.binaryStuff bool_nequals = (new union(.binaryStuff bit)
    # returns input1 != input2
    input1 = null
    input2 = null
    %deep c0 = (new union(.binaryStuff bit_nequals)
        input1 is {..input1 b0}
        input2 is {..input2 b0}
    )
    %deep c1 = (new union(.binaryStuff bit_nequals)
        prev is {..c0}
        input1 is {..input1 b1}
        input2 is {..input2 b1}
    )
    ...
    %deep c7 = (new union(.binaryStuff bit_nequals)
        prev is {..c6}
        input1 is {..input1 b7}
        input2 is {..input2 b7}
    )
    cLast is {.c7}
    b is {.cLast b or (.input1 negative xor
            (.input2 negative))}
);
```

Listing 7.23: Byte comparison construct using individual bit comparisons. Input bytes are given at the top and the result is the value of *b* which is indivisibly calculated by dependency.

where he states that there appear to be two major limitations. 1) the lack of iteration for generating large numbers of instances and 2) performance when more than 500k definitions are involved. The first problem can be resolved with richer support for meta-relations and/or richer forms of custom agency (cf. §§8.2.5 and 8.2.6). The second issue is a consequence of the Cadence tool as implemented in chapter 4 being only an unoptimised prototype. Whilst a calculator is a relatively small program, even if implemented from logic gate foundations, this model does show that Cadence supports flexible programs.

SCUBA Diving Model

A model of decompression when SCUBA diving was constructed by William Dangerfield which showed how a diver responds in certain scenarios (cf. figure 7.15). The model is another example of a construal that has evolved to become a program. The "program" can now be used visually and interactively to demonstrate to novice divers the problems of diving and surfacing too quickly[23].

The user of the model can make the diver dive to different depths at different rates and surface again. The model will then calculate nitrogen pressures in different parts of the body ("compartments") and display these to the user, highlighting them orange and red if they are too high. The model could, as Dangerfield comments in [Dangerfield, 2011], be extended to include other gas mixtures and additional compartments.

Whilst the model does, like the calculator, make extensive use of cloning, it is the use of dynamical definitions that is more significant here. The model is inherently dynamical in nature with gas concentrations continually changing over time without any particular reference to agency causing this change.

Dangerfield was not familiar with decompression models prior to starting the development process. He came to understand these models by experimentally and in-

[23] Dangerfield has shown the model to instructors who have commented that it would be useful.

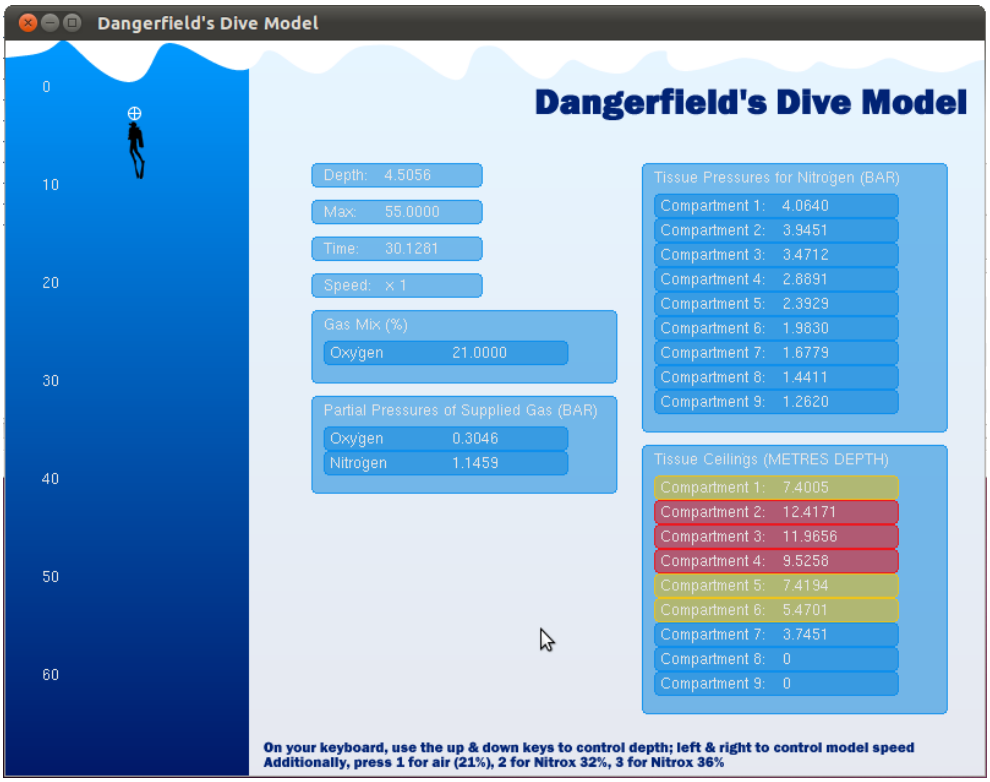

Figure 7.15: William Dangerfield's SCUBA Diving Model

crementally developing this diving model. In other words, the model started life as a provisional and private learning exercise which solidified into the more generic and public artefact presented in figure 7.15.

7.3.4 Student Feedback

Some feedback was gathered from the students after they had completed the coursework by using a web form questionnaire (cf. appendix C). For the most part the students say they understood the concepts of "computation by navigation" and "dynamical relations" (56% and 89% respectively replied yes when asked). The "friendliness" of the interface and DASM notation were also judged to be average to good, although 22% did suggest that DASM could be simpler and the comments relating to this were about confusion over the use of brackets and operator precedence.

For the students who did not choose to use Cadence for their coursework assignment, they were asked why. The response was overwhelmingly (56%) that there was not enough documentation and a lack of confidence in the tool (22%). The lack of confidence was often associated with a lack of documentation and example models[24].

Other criticisms and suggestions include the following:

- A lack of formal technical syntax (for DASM).
- Better (and earlier) explanation of navigating a tree (graph).
- Better file handling[25].
- A means of clearing state and variables to reset a model.
- Interface was unstable and crashed regularly.
- Performance in updating observables in larger models.

[24] Specifically it was the lack of documentation of the game library and %deep cloning mechanisms that were problematic

[25] The file handling facilities have since been improved by searching in customisable default locations.

- Missing math functions and no way to specify them[26]

Largely these problems are due to the experimental and prototype nature of the Cadence tool they were using. Documentation had not been written as the tool itself was still being developed. The other issues such as "clearing state" and "missing math functions" have already been identified in chapter 6 as needing further work and are discussed in chapter 8. The success of the models of those students who did choose to use Cadence is without question. Compare, for example, the calculator and SCUBA diving models with previous EM models in EDEN that have similar characteristics (e.g. Timetable [Beynon et al., 2000b; Keer, 2010]). The internal structure of the model is much more clearly and elegantly captured, and the interface to the models benefits from access to observables through the Cadence GUI and much superior visualisation of current state.

[26] Generic definitions were not shown to them, but nor are they sufficiently well implemented.

Chapter 8

Conclusions and Further Work

At the beginning of this thesis the question of "*how to bring extreme plasticity to software in a way suitable for everyday users?*" was asked and framed with reference to using the Empirical Modelling framework as a starting point. A more specific question asked how EM concepts and tools could be improved to support the notion of *plastic applications*. It was argued that EM concepts and tools were not at that time capable of scaling up to supporting plastic applications. The title of this thesis states the problem concisely, EM is a framework for modelling construals (on a computer) but it does not give adequate support for migrating from informal construals to formalisable programs without a translation step. So the goal of the work was to see if enhancing support for both construals and programs in a single environment allowed for such a migration. This migration is then an approach to enabling extreme plasticity in software.

First, in chapter 2, a brief summary of EM was given along with existing End-User Development research which is perhaps the closest research area in attempting to support end-user developed plastic applications. The notion of *refinement* from experience to program was introduced in chapter 2, along with various *dimensions* that have guided this work. The principles of EUD identified here should also be followed in the development of any new EUD environment. Chapter 3 then identified specific problems with the way current EM tools had been implemented with regards to supporting

programs and scaling up to realistic applications, but also with regards to supporting construals. From this a few specific suggestions for improvements were given which would ameliorate or resolve the identified issues. These suggestions were based upon existing technologies that had proven successful for traditional software in achieving increased flexibility. They were: 1) to semi-structure the observable dependency network and 2) to allow for dynamical dependencies. Following from these suggested improvements a new tool (Cadence) was developed which aimed to maintain the EUD principles and stick with the ODA framework of EM whilst implementing the improvements. Many examples were then given to show how Cadence was still able to support Empirical Modelling but also how the new concepts could radically improve upon existing EM models enabled by EDEN.

Having developed Cadence the thesis then goes on to discuss a new framework based upon an idealisation of the prototype tool. The framework gives an account of Cadence that is both informal and formal in order to provide both an informal and formal semantics. Such an account is needed to show how Cadence can support both construals and programs. A discussion is then given of how Cadence does improve support for both construals and programs in comparison to EDEN. Finally, chapter 7 of the thesis goes back to Empirical Modelling to demonstrate that Cadence concepts need to be fundamental in any EM tool in order to take full advantage of them. It also highlights further limitations of the Cadence implementation of chapter 4. Student coursework models are then used to show that Cadence does support the migration from construal to program, much more so than existing tools.

There are many limitations with this work, the primary one being the scale and complexity of the resulting plastic applications. However, the aim of the thesis was not to show large scale and complex applications but to show a way of supporting end-user developed applications that start life as construals and migrate to being considered as programs. The aim has been to improve EM tool support with this objective in mind. As Beynon states at the end of Beynon [2011]: "[Imagine] what might be achieved by

investing as much effort in developing effective tools for developing construals as has been dedicated to tools for developing programs". If a true Cadence environment were to be implemented, as an operating system, then it could radically alter the way in which computers are used by the everyday user.

8.1 Contributions

The work contained within this thesis has made several contributions to both Empirical Modelling research and to the broader computer science community. Specific contributions are summarised below:

- The development of a new tool for Empirical Modelling and programming. The tool has already proven useful in the teaching of Empirical Modelling and has much potential to be improved to the point where it could become the primary EM tool. Outside of EM the tool demonstrates how EM concepts can be applied to more traditional problems in managing software, where dependency can be used as *glue* between components, as shown in the Stargate model.

- Enhancing the EM conceptual framework by removing the focus from EDEN. Previously the EDEN implementation of EM has had an unjustified hidden impact upon the conceptual framework, particularly relating to the nature of observables and dependency. Whilst the ODA concepts themselves remain the same, the interpretations of them in EDEN terms have led to much confusion. For example: the role of functions, notations, time and structures (moding issues). The discussion of the Cadence framework in chapter 6 has highlighted these issues by reinterpreting the EM framework in Cadence terms which reveals the original bias.

- Improved support for construals on computers. The addition of structural relations has allowed for the removal of types and specific algebras from the underlying representation of construals. As a consequence of this work it is now possible to

represent richer and more varied kinds of construal than were previously possible with EDEN. The inflexibility of definitive notations has been identified as a problem and somewhat resolved through the Cadence framework.

- Demonstrating that a transition from informal modelling of construals to formalisable programs is in principle possible using the EM framework, without a translation step. This has been illustrated with various models developed in Cadence both by the author of this thesis and other students.

- Empirical Modelling has in the past been somewhat distanced from more traditional software development processes and technologies. The discussion of EM given in this thesis, along with the updating of its tools using recent technologies helps to relate EM to the traditional software world. In doing this the community at large can better understand the purpose and use of EM thinking which has previously been rather difficult to get across.

- The ideas initially explored by Subtext and Forms/3 have been taken further in this thesis. Whilst Subtext was a practical attempt at becoming less paper-centric by programming a tree structure that contains what we term dependency, Cadence has looked more deeply at the conceptual issues by linking with Empirical Modelling and also at scaling up the use of similar structures and dependency. The thesis has further demonstrated the potential of the Subtext concept through far richer examples, tools and concepts. Similarly, Cadence also contains the notion of temporal formula found in Forms/3 which has been used extensively in relatively complex models.

8.2 Further Work

This thesis introduces new tools and concepts which, as has been made clear throughout the thesis, require much further work. A brief summary of the major areas of further work already highlighted is given here. In some cases considerable work has already been

done by the author in these areas that has not been elaborated upon in the thesis. In others the need for further work reflects time and resource constraints.

8.2.1 Larger Project and Complex Applications

All the examples given within this thesis have been relatively small scale applications. Whilst the scale of the applications is still considerably larger than that supported by EDEN, it still falls far short of many real world applications. To a large extent the reason for this is the prototype nature of the Cadence tool developed in this thesis which is too unstable and lacking in features to scale appropriately. These issues are discussed as other forms of further work, but there will be a need to see just how complex and large applications within a Cadence-like environment can be.

8.2.2 Interactive Development Environment

In §6.1.1 a need to move away from using DASM scripts was identified, with the suggestion being to take a more visual approach. Taking a visual approach to manipulating the Cadence OD-net matches well with the EUD visual programming attempts and the need for *directness*. The existing interface was one area that was criticised by students using Cadence with some of them suggesting possible improvements.

The interface currently includes a tree representation of the OD-net which is somewhat inadequate for representing a graph. Ideas have been explored for generic ways of representing and exploring objects besides the use of a tree. One other consideration is that any particular structure can be interpreted in many ways and so it should be possible to interact with and visualise a structure differently through applying different mediator agents. The direct manipulation of Forms/3 is one approach. The style of programming in Subtext is another (cf. §2.1.3). Recent work by Abi-Antoun et al. on visualising runtime object structure is also particularly relevant as a way of visualising the OD-net [Abi-Antoun and Selitsky, 2010]. Self with its Morphic interface can also provide inspiration for ways of developing an OD-net.

8.2.3 Collaboration and Distribution

A vision for Cadence is for the OD-net to be distributed globally so that dependencies and relationships could exist between observables all over the world. An early version of this vision involved being able to share hardware devices across machines in a transparent manner. For example, the mouse on one machine could by dependency be linked to the mouse on another to allow for remote control. Such connections could be set up with ease. There has been a suggestion of global distribution with EM previously in connection with F-Rep shape modelling [Cartwright et al., 2005], something that may well benefit from using Cadence.

Network distribution was explored in §5.3. To scale up and work efficiently it will be necessary to develop caching mechanisms, scheduling algorithms, security and a global OID allocation mechanism. Beyond these technical concerns it will also be vital to see how construals and applications can be refined collaboratively by potentially hundreds or thousands of individuals. It may be appropriate to regard the entire globally distributed OD-net as a single system in which case there would potentially be billions of people involved in constructing it. Without doubt, this would need substantial further work and may not even be realistic.

8.2.4 Histories and Persistence

Although this has not been documented within the main body of this thesis, there has been considerable effort made in finding a way to make the Cadence OD-net persistent between sessions. This involves finding an effective way of storing the entire OD-net to disk to be reloaded when the environment is restarted. Related to this is the possibility of caching parts of the OD-net in memory whilst leaving the rest on disk so that larger nets may be supported. The benefit of a persistent OD-net is that reliance on DASM scripts could be reduced or removed entirely (subject to developing appropriate alternative interfaces).

Various compression and caching mechanisms were developed but not fully im-

plemented in the Cadence prototype. One particular concern is the ability to undo damaging changes to a model effectively. While in EDEN undoing changes involves reverting a definition to its previous form, in Cadence - due to dynamical definitions - there may be more to undo. A form of version control of OD-net history would be vital in achieving this since a persistent net cannot be reverted to an original state as easily as an OD-net constructed from scripts.

Some of the compression ideas included the automatic classifications of structures in the OD-net to move content descriptions common to many instances into classes. Such mechanisms would be hidden from the user who would still be completely free to change structures. Also, history data can be stored as it is in classic version control, by storing only the differences. In Cadence it would be adequate to store only agent changes since every other change could be recalculated. Storing the occasional snapshot would help improve performance.

8.2.5 Custom Agency: Security and Schema

At present there is little support for custom agency as can be found in EDEN. Agency in Cadence currently requires the modeller to write C++ modules which reduces the liveness of the environment. There has been one attempt to remedy this by providing simple conditional actors that can perform single assignments when a condition is satisfied. These actors can be chained together to act sequentially or they can act in parallel to make changes. This solution is not particularly elegant or sufficiently rich for many tasks (i.e. looping to generate large numbers of new definitions). The hybrid described in §7.2 was constructed to overcome this lack of agency.

In addition to richer custom agency, an account of security and other forms of restriction is required. There has been some suggestion previously in the thesis that capability-based security would be ideally suited for use in the OD-net. Agents have capabilities based upon the kind of OID they have for manipulating a particular object. This approach would allow for distributed security in a collaborative and distributed

environment, as well as providing a mechanism for restricting agent actions to help solidify the artefact into a program. No work has yet been done on implementing such a security mechanism.

8.2.6 Support for Meta-Relations

As pointed out when discussing latent relations in §6.1.2, the Cadence prototype only supports single variable meta-dependency. A story for how meta-relations relate to programs and construals needs to be developed further than it has in this thesis before deciding whether Cadence should support multi-variable meta-dependency and meta-structure. How to implement an environment that does support the full range of OD-net meta-relations is unclear as it would be, or so it appears at the moment, rather complex to mix with the more concrete enumeration approach currently supported.

With full support for meta-relations it would be possible to develop a comprehensive library of generic functions for use in Cadence. Currently it is difficult to specify many kinds of functions without resorting to oracles, which limits the kinds of models that Cadence can practically implement. The ability to develop generic functions would be vital in scaling up to more substantial construals and programs.

8.2.7 Optimisations and Concurrency

The implementation of Cadence described in chapter 4 has undergone many revisions and has been developed to allow for experimentation with its architecture. As a consequence in many respects it is far from an optimised implementation. Specifically the storage of objects and representations used for definitions are entirely unoptimised which leads to increased use of memory and reduced execution efficiency. Techniques such as automatic classification of structures could reduce the memory footprint of a model as well as improve lookup performance. Definitions could be JIT compiled in ways that are similar to how methods are optimised in Self [Hölzle and Ungar, 1994]. Despite the lack of optimisation it still performs substantially better than EDEN (cf. Calculator model in

§7.3.3).

It is also conceivable that definitions could be updated in parallel. The existing implementation did at one time support the concurrent processing of events which did increase performance on multi-core machines[1]. However, the implementation of this concurrency was poor and there are synchronisation issues which need further exploration.

8.2.8 Formal Account

Finally, the formal account of Cadence given in chapter 6 is only a provisional first attempt that has not been rigorously critiqued or put to use. There is a need to show the universality of Cadence and elaborate on agency as well as relating it to other concepts. Being able to prove things about Cadence artefacts would go a long way towards it being used for serious and critical applications, which would perhaps be necessary if Cadence were to be considered as an operating system.

8.3 Limitations of the Approach

Beyond any practical limitations due to time constraints, there are also certain areas where this approach to plastic applications is not appropriate. Any applications that are to be used in safety critical situations may require a degree of formal proof that is perhaps not achievable with the framework discussed in this thesis. In which case a translation step might be required, going via a more formal specification. Alternatively a specification may be developed to which a Cadence model must be matched.

Although the potential for just-in-time (JIT) optimisation has been touched upon in this thesis, it is still unlikely that performance would be good enough for the most demanding applications. With high-performance applications a translation step may also be required to optimise for specific hardware. The problem here is that hardware is

[1] Performance improved by around 50% on dual core machines for models with large numbers of definitions.

typically imperative in character and so imperative programs are better suited to it. If specific hardware were to be developed for Cadence then such performance issues may become less sigificant[2].

Finally, many existing problems, algorithms and applications can be adequately developed using traditional programs. There is little benefit in using the framework in this thesis to develop these. Instead problems of a pre-theoretical nature or those involving end-user development would benefit the most.

8.4 Looking Forward

There has been considerable focus in this thesis on Empirical Modelling. However, the original and future vision for Cadence is largely independent of Empirical Modelling, with the intention being to develop it for more widespread use. Empirical Modelling has been used here to provide an underlying framework for Cadence, but with this now largely in place it is possible to focus on more practical concerns such as network distribution, concurrency, persistence and moving towards the plastic operating system vision stated in chapter one. Recent developments by Google, for example with their Chrome OS, are in many respects remarkably similar to how Cadence as an operating system would function. Cloud computing, where almost everything is on-line, is increasingly becoming a reality and there is a need for better operating system support (moving away from Linux). Various problems that Chrome OS suffers from could be dealt with elegantly by Cadence if some of the further work suggested above is carried out. In particular the transparency of a shared OD-net, caching potential and use of dependency could overcome limitations such as dependence on an internet connection for applications in Chrome OS. The richness and ease with which Cadence applications can be developed, adapted and potentially transparently shared matches well with these recent developments by Google, Apple and Microsoft.

[2]There has been some consideration of developing specific hardware for Cadence but it remains unknown as to whether this is possible or practical.

Appendix A

DASM Scripts for Models

A.1 Stargate Scripts

```
%include "wgd/wgd.dasm";

#@sgwidget = (this widgets root);
@sgwidget = (new union (@prototypes window)
    x = 10
    y = 10
    width = 640
    height = 480
);
#.widgets root children sgate = (@sgwidget);

%include "stargate/window.dasm";
%include "stargate/viz.dasm";
%include "stargate/gate.dasm";

@stargate id = stargate;
@ui browser root = (@stargate);
```

Listing A.1: stargate.dasm

```
#this widgets root children stargate = (this prototypes window _clone);
@mouse grab := { .buttons left };
this bloom = (@bloom = (new)

    blur = (new
        type = Shader
        vert = (new type=LocalFile filename="stargate/data/blur.vp")
        frag = (new type=LocalFile filename="stargate/data/blur.fp")
```

```
9       )
10
11      bloom = (new
12          type = Shader
13          vert = (new type=LocalFile
14                   filename="stargate/data/extractBloom.vp")
15          frag = (new type=LocalFile
16                   filename="stargate/data/extractBloom.fp")
17      )
18
19      tone = (new
20          type = Shader
21          vert = (new type=LocalFile filename="stargate/data/tone.vp")
22          frag = (new type=LocalFile filename="stargate/data/tone.fp")
23      )
24
25      0 = (new
26          type=RenderTarget
27          depth = true
28          scene = (new
29              type = Scene3D
30              instances = (new)
31          )
32
33          camera = (@camera = (new)
34              type = Camera3D
35              fov = 60.0
36              near = 1.0
37              far = 100.0
38
39              up := { @keyboard keys up }
40              left := { @keyboard keys left }
41              down := { @keyboard keys down }
42              right := { @keyboard keys right }
43
44              dup := { if (.up) {-1.6} else {0.0} }
45              dleft := { if (.left) {1.6} else {0.0} }
46              ddown := { if (.down) {1.6} else {0.0} }
47              dright := { if (.right) {-1.6} else {0.0} }
48
49              scalexz := {
50                  @math cos (.orientation x)
51              }
52
53              deltax := {
54                  @math sin (.orientation y - 1.5707) * (.dleft +
55                  (.dright)) + (@math cos (.orientation y - 1.5707) *
56                  (.dup + (.ddown)) * (.scalexz))
57              }
58
59              deltay := {
60                  @math sin (.orientation x) * (.dup + (.ddown)) * -1.0
61              }
62
63              deltaz := {
```

```
64                  @math sin (.orientation y) * (.dleft + (.dright)) +
65                  (@math cos (.orientation y) * (.dup + (.ddown)) *
66                  (.scalexz))
67              }
68
69
70              position = (new x = 0.0 y = 0.0 z = 3.0
71                  x := { .x + (@root itime * (@camera deltax)) }
72                  y := { .y + (@root itime * (@camera deltay)) }
73                  z := { .z + (@root itime * (@camera deltaz)) }
74              )
75
76              mousex := { if(@mouse grab) { @mouse deltax } else {0} }
77              mousey := { if(@mouse grab) { @mouse deltay } else {0} }
78
79              orientation = (new x = 0.0 y = 0.0 z = 0.0
80                  x := { .x - (@root itime * (@camera mousey)) }
81                  y := { .y - (@root itime * (@camera mousex)) }
82              )
83          )
84
85          texture = (new
86              type = Texture
87              width is { @sgwidget children cam1 width }
88              height is { @sgwidget children cam1 height }
89              compress = false
90              hdr = true
91              nearest = true
92              clamp = true
93          )
94
95          depthtexture = (new
96              type = Texture
97              hdr = false
98              nearest = false
99              compress = false
100             clamp = true
101         )
102     )
103
104     12 = (new
105         type=RenderTarget
106
107         source := { @bloom 0 }
108         material = (new
109             type = Material
110             shader := { @bloom blur }
111             #variables go here.
112             variables = (new
113                 dx := { 1.0 / (@bloom 0 texture width) }
114                 dy = 0.0
115                 #tex = 0
116             )
117         )
118
```

```
119             texture = (new
120                 type = Texture
121                 width := { @bloom 0 texture width }
122                 height := { @bloom 0 texture height }
123                 hdr = true
124                 clamp = true
125             )
126         )
127 
128         13 = (new
129             type=RenderTarget
130 
131             source := { @bloom 12 }
132             material = (new
133                 type = Material
134                 shader := { @bloom blur }
135                 #variables go here.
136                 variables = (new
137                     dx = 0.0
138                     dy := { 1.0 / (@bloom 0 texture height) }
139                     #tex = 0
140                 )
141             )
142 
143             texture = (new
144                 type = Texture
145                 width := { @bloom 0 texture width }
146                 height := { @bloom 0 texture height }
147                 hdr = true
148                 clamp = true
149             )
150         )
151 
152         14 = (new
153             type=RenderTarget
154 
155             source := { @bloom 13 }
156             material = (new
157                 type = Material
158                 shader := { @bloom blur }
159                 #variables go here.
160                 variables = (new
161                     dx := { 1.0 / (@bloom 14 texture width) }
162                     dy = 0.0
163                     #tex = 0
164                 )
165             )
166 
167             texture = (new
168                 type = Texture
169                 width := { @bloom 13 texture width / 4 }
170                 height := { @bloom 13 texture height / 4 }
171                 hdr = true
172                 clamp = true
173             )
```

```
174         )
175
176         15 = (new
177             type=RenderTarget
178
179             source := { @bloom 14 }
180             material = (new
181                 type = Material
182                 shader := { @bloom blur }
183                 #variables go here.
184                 variables = (new
185                     dx = 0.0
186                     dy := { 1.0 / (@bloom 14 texture height) }
187                     #tex = 0
188                 )
189             )
190
191             texture = (new
192                 type = Texture
193                 width := { @bloom 13 texture width / 2 }
194                 height := { @bloom 13 texture height / 2 }
195                 hdr = true
196                 clamp = true
197             )
198         )
199
200         11 = (new
201             type=RenderTarget
202
203             source := { @bloom 13 }
204             material = (new
205                 type = Material
206                 shader = (new
207                     type=Shader
208                     vert = (new type=LocalFile
209                             filename="stargate/data/dof.vp")
210                     frag = (new type=LocalFile
211                             filename="stargate/data/dof.fp")
212                 )
213                 variables = (new
214                     btex = 0
215                     dtex = 1
216                     tex = 2
217                     #w := { @bloom 0 texture width div 2 }
218                     #h := { @bloom 0 texture height div 2 }
219                 )
220                 textures = (new
221                     1 := { @bloom 0 depthtexture }
222                     2 := { @bloom 0 texture }
223                 )
224             )
225
226             texture = (new
227                 type = Texture
228                 width := { @bloom 0 texture width }
```

```
229             height := { @bloom 0 texture height }
230             hdr = true
231             clamp = true
232         )
233     )
234 
235     1 = (new
236         type=RenderTarget
237 
238         source := { @bloom 0 }
239         material = (new
240             type = Material
241             shader = (new
242                 type = Shader
243                 vert = (new type=LocalFile
244                         filename="stargate/data/extractBloom.vp")
245                 frag = (new type=LocalFile
246                         filename="stargate/data/extractBloom.fp")
247             )
248             #variables go here.
249             variables = (new
250                 brightThreshold := {
251                     @bloom 10 material variables brightThreshold }
252                 #w = 300.0
253                 #h = 200.0
254                 #tex = 0
255             )
256         )
257 
258         texture = (new
259             type = Texture
260             width := { @bloom 0 texture width / 2 }
261             height := { @bloom 0 texture height / 2 }
262             hdr = true
263             clamp = true
264         )
265     )
266 
267     2 = (new
268         type=RenderTarget
269 
270         source := { @bloom 1 }
271         material = (new
272             type = Material
273             shader := { @bloom blur }
274             #variables go here.
275             variables = (new
276                 #w := { @bloom 2 texture width }
277                 #h := { @bloom 2 texture height }
278                 dx := { 1.0 / (@bloom 2 texture width) }
279                 dy = 0.0
280                 #tex = 0
281             )
282         )
283 
```

```
284             texture = (new
285                 type = Texture
286                 width := { @bloom 1 texture width / 4 }
287                 height := { @bloom 1 texture height / 4 }
288                 hdr = true
289                 clamp = true
290             )
291         )
292 
293         3 = (new
294             type=RenderTarget
295 
296             source := { @bloom 2 }
297             material = (new
298                 type = Material
299                 shader := { @bloom blur }
300                 #variables go here.
301                 variables = (new
302                     #w := { @bloom 3 texture width }
303                     #h := { @bloom 3 texture height }
304                     #tex = 0
305                     dy := { 1.0 / (@bloom 2 texture height) }
306                     dx = 0.0
307                 )
308             )
309 
310             texture = (new
311                 type = Texture
312                 width := { @bloom 1 texture width / 4 }
313                 height := { @bloom 1 texture height / 4 }
314                 hdr = true
315                 clamp = true
316             )
317         )
318 
319         4 = (new
320             type=RenderTarget
321 
322             source := { @bloom 3 }
323             material = (new
324                 type = Material
325                 shader := { @bloom blur }
326                 #variables go here.
327                 variables = (new
328                     #w := { @bloom 2 texture width }
329                     #h := { @bloom 2 texture height }
330                     dx := { 1.0 / (@bloom 4 texture width) }
331                     dy = 0.0
332                     #tex = 0
333                 )
334             )
335 
336             texture = (new
337                 type = Texture
338                 width := { @bloom 3 texture width / 4 }
```

```
339             height := { @bloom 3 texture height / 4 }
340             hdr = true
341             clamp = true
342         )
343     )
344 
345     5 = (new
346         type=RenderTarget
347 
348         source := { @bloom 4 }
349         material = (new
350             type = Material
351             shader := { @bloom blur }
352             #variables go here.
353             variables = (new
354                 #w := { @bloom 3 texture width }
355                 #h := { @bloom 3 texture height }
356                 #tex = 0
357                 dy := { 1.0 / (@bloom 4 texture height) }
358                 dx = 0.0
359             )
360         )
361 
362         texture = (new
363             type = Texture
364             width := { @bloom 3 texture width / 4 }
365             height := { @bloom 3 texture height / 4 }
366             hdr = true
367             clamp = true
368         )
369     )
370 
371     6 = (new
372         type=RenderTarget
373 
374         source := { @bloom 5 }
375         material = (new
376             type = Material
377             blending = one
378             #shader := { @bloom blur }
379             #variables go here.
380             #variables = (new
381             #   w = 300.0
382             #   h = 200.0
383                 #tex = 0
384             #)
385         )
386 
387         clear = false
388         texture := { @bloom 3 texture }
389     )
390 
391     7 = (new
392         type=RenderTarget
393 
```

```
394             source := { @bloom 6 }
395             material = (new
396                 type = Material
397                 blending = one
398                 #shader := { @bloom blur }
399                 #variables go here.
400                 #variables = (new
401                 #   w = 300.0
402                 #   h = 200.0
403                     #tex = 0
404                 #)
405             )
406 
407             clear = false
408             texture := { @bloom 3 texture }
409         )
410 
411         8 = (new
412             type=RenderTarget
413 
414             source := { @bloom 7 }
415             material = (new
416                 type = Material
417                 blending = one
418                 #shader := { @bloom blur }
419                 #variables go here.
420                 #variables = (new
421                 #   w = 300.0
422                 #   h = 200.0
423                     #tex = 0
424                 #)
425             )
426 
427             clear = false
428             texture := { @bloom 2 texture }
429         )
430 
431         9 = (new
432             type=RenderTarget
433 
434             source := { @bloom 6 }
435             material = (new
436                 type = Material
437                 blending = one
438                 #shader := { @bloom blur }
439                 #variables go here.
440                 #variables = (new
441                 #   dx := { 1.0 div (@bloom 3 texture width) }
442                 #   dy = 0.0
443                 #   tex = 0
444                 #)
445             )
446 
447             clear = false
448             texture := { @bloom 1 texture }
```

```
449         )
450
451         10 = (new
452             type=RenderTarget
453
454             source := { @bloom 9 }
455             material = (new
456                 type = Material
457                 #blending = one
458                 shader := { @bloom tone }
459                 variables = (new
460                     brightThreshold = 0.9
461                     bloomFactor = 0.4
462                     exposure = 0.7
463                     tex = 0
464                     bloom = 1
465                 )
466
467                 textures = (new
468                     0 := { @bloom 0 texture }
469                     1 := { @bloom 9 texture }
470                 )
471             )
472
473             #clear = false
474             #texture := { @bloom 0 texture }
475             texture = (new
476                 type = Texture
477                 width := { @bloom 0 texture width }
478                 height := { @bloom 0 texture height }
479                 compress = false
480                 hdr = true
481                 clamp = true
482             )
483         )
484     );
485
486     #this widgets root children stargate title = "Stargate Game";
487     #this widgets root children stargate height = 750;
488     #this widgets root children stargate width = 1000;
489     #this widgets root children #stargate children
490     @sgwidget children cam1 = (new
491         type = WViewport
492         x = 0
493         y = 20
494         #parent = (this widgets root children stargate)
495         width := { @sgwidget width }
496         height := { @sgwidget height - 20 }
497         visible = true
498
499         source = (this bloom 10)
500
501         #scene := { @bloom 0 scene }
502         #camera := { @bloom 0 camera }
503
```

```
504         children = (new)
505     );
506
507     @sgwidget children cam1 children dial =
508         (new union (@prototypes button));
509     @sgwidget children cam1 children dial
510         caption = "Dial"
511         visible = true
512         x = 10
513         y = 30
514         width = 100
515         #dial_agent := { if (this mousedown) {
516               @root sgobjects stargate dial = true } }
517         height = 20;
```

Listing A.2: window.dasm

```
1
2   this chevronshader = (new
3       type = Shader
4       vert = (new type=LocalFile filename="stargate/data/chevron.vert")
5       frag = (new type=LocalFile filename="stargate/data/chevron.frag")
6   );
7
8   this chevrontextures = (new
9       0 = (new
10          type = Texture
11          file = (new
12              type = LocalFile
13              filename = "stargate/data/chevron_diffuse.tga"
14          )
15      )
16      1 = (new
17          type = Texture
18          file = (new
19              type = LocalFile
20              filename = "stargate/data/chevron_normal.tga"
21          )
22      )
23  );
24
25  this sgobjects = (this bloom 0 scene instances);
26  this sgobjects stargate = (@stargate = (new type = IModel));
27  this sgobjects puddle = (new type = IPrimitive3D);
28
29  this sgobjects puddle material = (new
30          type = Material
31          shader = (new
32              type = Shader
33              vert = (new type=LocalFile
34                      filename="stargate/data/puddle.vert")
35              frag = (new type=LocalFile
36                      filename="stargate/data/puddle.frag")
```

```
37          )
38          variables = (new
39              world = 0
40              hole = 1.0
41              hole := { @stargate hole }
42              time := { @root time * 4.0 }
43              sd = 0.3
44              bright = 0.3
45          )
46
47          textures = (new
48              0 = (this widgets root sprite texture)
49          )
50      )
51  ;
52
53  this sgobjects stargate model = (new
54          type = Model
55          file = (new type=LocalFile
56                  filename="stargate/data/stargate.3ds")
57
58          materials = (new
59              Chevron0 = (new
60                  type = Material
61                  shininess = 2
62                  shader = (this chevronshader)
63                  textures = (this chevrontextures)
64                  variables = (new
65                      colourMap = 0
66                      normalMap = 1
67                      on := { @stargate chevrons 0 on }
68                      position := { @stargate chevrons 0 position }
69                  )
70                  diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
71                  ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
72                  specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
73              )
74
75              Chevron1 = (new
76                  type = Material
77                  shininess = 2
78                  shader = (this chevronshader)
79                  textures = (this chevrontextures)
80                  variables = (new
81                      colourMap = 0
82                      normalMap = 1
83                      on := { @stargate chevrons 1 on }
84                      position := { @stargate chevrons 1 position }
85                  )
86                  diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
87                  ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
88                  specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
89              )
90
91              Chevron2 = (new
```

```
92                  type = Material
93                  shininess = 2
94                  shader = (this chevronshader)
95                  diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
96                  ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
97                  specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
98                  textures = (this chevrontextures)
99                  variables = (new
100                     colourMap = 0
101                     normalMap = 1
102                     on := { @stargate chevrons 2 on }
103                     position := { @stargate chevrons 2 position }
104                 )
105             )
106
107             Chevron3 = (new
108                 type = Material
109                 shininess = 2
110                 shader = (this chevronshader)
111                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
112                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
113                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
114                 textures = (this chevrontextures)
115                 variables = (new
116                     colourMap = 0
117                     normalMap = 1
118                     on := { @stargate chevrons 3 on }
119                     position := { @stargate chevrons 3 position }
120                 )
121             )
122
123             Chevron4 = (new
124                 type = Material
125                 shininess = 2
126                 shader = (this chevronshader)
127                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
128                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
129                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
130                 textures = (this chevrontextures)
131                 variables = (new
132                     colourMap = 0
133                     normalMap = 1
134                     on = 0
135                     position = 0.0
136                 )
137             )
138
139             Chevron5 = (new
140                 type = Material
141                 shininess = 2
142                 shader = (this chevronshader)
143                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
144                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
145                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
146                 textures = (this chevrontextures)
```

```
147                 variables = (new
148                     colourMap = 0
149                     normalMap = 1
150                     on = 0
151                     position = 0.0
152                 )
153             )
154 
155             Chevron6 = (new
156                 type = Material
157                 shininess = 2
158                 shader = (this chevronshader)
159                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
160                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
161                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
162                 textures = (this chevrontextures)
163                 variables = (new
164                     colourMap = 0
165                     normalMap = 1
166                     on := { @stargate chevrons 6 on }
167                     position := { @stargate chevrons 6 position }
168                 )
169             )
170 
171             Chevron7 = (new
172                 type = Material
173                 shininess = 2
174                 shader = (this chevronshader)
175                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
176                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
177                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
178                 textures = (this chevrontextures)
179                 variables = (new
180                     colourMap = 0
181                     normalMap = 1
182                     on := { @stargate chevrons 7 on }
183                     position := { @stargate chevrons 7 position }
184                 )
185             )
186 
187             Chevron8 = (new
188                 type = Material
189                 shininess = 2
190                 shader = (this chevronshader)
191                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
192                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
193                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
194                 textures = (this chevrontextures)
195                 variables = (new
196                     colourMap = 0
197                     normalMap = 1
198                     on := { @stargate chevrons 8 on }
199                     position := { @stargate chevrons 8 position }
200                 )
201             )
```

```
202
203             Default = (new
204                 type = Material
205                 shininess = 2
206                 shader = (new
207                     type = Shader
208                     vert = (new type=LocalFile
209                             filename="stargate/data/normal.vert")
210                     frag = (new type=LocalFile
211                             filename="stargate/data/normal.frag")
212                 )
213
214                 textures = (new
215                     0 = (new
216                         type = Texture
217                         file = (new
218                             type = LocalFile
219                             filename = "stargate/data/rim_diffuse.tga"
220                         )
221                     )
222                     1 = (new
223                         type = Texture
224                         file = (new
225                             type = LocalFile
226                             filename = "stargate/data/rim_normal.tga"
227                         )
228                     )
229                 )
230
231                 variables = (new
232                     colourMap = 0
233                     normalMap = 1
234                 )
235
236                 diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
237                 ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
238                 specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
239             )
240
241             Symbols = (new
242                 type = Material
243                 shininess = 2
244                 shader = (new
245                     type = Shader
246                     vert = (new type=LocalFile
247                             filename="stargate/data/ring.vert")
248                     frag = (new type=LocalFile
249                             filename="stargate/data/normal.frag")
250                 )
251
252                 textures = (new
253                     0 = (new
254                         type = Texture
255                         file = (new
256                             type = LocalFile
```

```
                                filename =
                                    "stargate/data/symbol_diffuse.tga"
                            )
                        )
                        1 = (new
                            type = Texture
                            file = (new
                                type = LocalFile
                                filename =
                                    "stargate/data/symbol_normal.tga"
                            )
                        )
                    )

                    variables = (new
                        colourMap = 0
                        normalMap = 1
                        angle := { @stargate rotation }
                    )

                    diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
                    ambient = (new r=1.0 g=1.0 b=1.0 a=1.0)
                    specular = (new r=0.7 g=0.7 b=0.7 a=1.0)
                )
            )
        );
```

Listing A.3: viz.dasm

```
this sgobjects stargate

    symbols = (new
        0 = 0
        1 = 0
        2 = 0
        3 = 0
        4 = 0
        5 = 0
        6 = 0
    )

    dial = false
    dial := { if (@sgwidget children cam1 children dial mousedown)
            {true} else {..dial} }

    cursym = 0
    prevsym := {.cursym}
    cursym := {
        if (.dial) {
            if (..locked and (..prevsym == (..cursym))) {
                ..cursym + 1
            } else {
                ..cursym
```

```
25              }
26          } else {0}
27      }
28
29      match = false
30      ready = false
31      locked = false
32      rotspeed = -0.3
33      rotation = 0.0
34      rotation := {
35          if (.dial and (.match not or (.locked)) and (.ready not)) {
36              ..rotation + (..rotspeed * (@root itime))
37          } else {
38              ..rotation
39          }
40      }
41
42      #match := { this cursym; false }
43      symrot is { 0.1611 * (.symbols (.cursym)) - (0.6981 * (.cursym)) }
44      match is { .rotation < (.symrot + 0.1) and (.rotation >
45              (.symrot - 0.1)) and (.dial) }
46
47      chevspeed = 0.8
48      chevmove = 0.0
49      chevdir = false
50
51
52      #Why does this work and the other not!
53      #chevif = (new
54      #    nif = (new
55      #        true is {true}
56      #        false is {...chevdir}
57      #    )
58      #    true is { false }
59      #    false is { .nif (..chevmove > 0.9999) }
60      #)
61      #chevdir := { .chevif (.match == false) }
62
63      chevdir := {
64          if (.match == false) {false} else {
65              if (..chevmove > 0.9999) {true} else {
66                  ..chevdir
67              }
68          }
69      }
70
71      chevmove := {
72          if (.match) {
73              if (..locked) {0.0} else {
74                  if (..chevdir) {
75                      ..chevmove - (..chevspeed * (@root itime))
76                  } else {
77                      ..chevmove + (..chevspeed * (@root itime))
78                  }
79              }
```

```
        } else {0.0}
    }

    chevrons = (new
        0 = (new
            on = 0
            on := {
                if (@stargate cursym == 0) {
                    if (@stargate locked) {1} else {0}
                } else {
                    ..on
                }
            }
            position = 0.0
            position := {
                if (@stargate cursym == 0) {
                    @stargate chevmove
                } else {
                    ..position
                }
            }
        )

        1 = (new
            on = 0
            on := {
                if (@stargate cursym == 1) {
                    if (@stargate locked) {1} else {0}
                } else {
                    ..on
                }
            }
            position = 0.0
            position := {
                if (@stargate cursym == 1) {
                    @stargate chevmove
                } else {
                    ..position
                }
            }
        )

        2 = (new
            on = 0
            on := {
                if (@stargate cursym == 2) {
                    if (@stargate locked) {1} else {0}
                } else {
                    ..on
                }
            }
            position = 0.0
            position := {
                if (@stargate cursym == 2) {
                    @stargate chevmove
```

```
                } else {
                    ..position
                }
            }
        )

        3 = (new
            on = 0
            on := {
                if (@stargate cursym == 3) {
                    if (@stargate locked) {1} else {0}
                } else {
                    ..on
                }
            }
            position = 0.0
            position := {
                if (@stargate cursym == 3) {
                    @stargate chevmove
                } else {
                    ..position
                }
            }
        )

        6 = (new
            on = 0
            on := {
                if (@stargate cursym == 4) {
                    if (@stargate locked) {1} else {0}
                } else {
                    ..on
                }
            }
            position = 0.0
            position := {
                if (@stargate cursym == 4) {
                    @stargate chevmove
                } else {
                    ..position
                }
            }
        )

        7 = (new
            on = 0
            on := {
                if (@stargate cursym == 5) {
                    if (@stargate locked) {1} else {0}
                } else {
                    ..on
                }
            }
            position = 0.0
            position := {
```

```
                    if (@stargate cursym == 5) {
                        @stargate chevmove
                    } else {
                        ..position
                    }
                }
            )

            8 = (new
                on = 0
                on := {
                    if (@stargate cursym == 6) {
                        if (@stargate locked) {1} else {0}
                    } else {
                        ..on
                    }
                }
                position = 0.0
                position := {
                    if (@stargate cursym == 6) {
                        @stargate chevmove
                    } else {
                        ..position
                    }
                }
            )
        )

        locked is { .chevmove < 0.0001 and (.chevdir == true) }

        ready is { .cursym == 7 }
        #ready = true

        holespeed = 0.8
        hole = 1.0
        active is { .hole < -0.9999 }
        hole := {
            if (.ready) {
                if (..active) {-1.0} else {
                    ..hole - (..holespeed * (@root itime))
                }
            } else {1.0}
        }


        position = (new x=0.0 y=0.0 z=-4.0)

        rotspeed2 = 0.0
        orientation = (new x=0.0 y=-0.5 z=0.0
            y := { .y + (@root itime * (@stargate rotspeed2)) }
        )

        visible = true
    ;

```

```
this sgobjects puddle

    primitive = cube
    width = 3.3
    height = 3.3
    depth = 0.1
    visible = true

    position = (new x=0.0 y=0.0 z=-3.0
        z := { @stargate position z }
    )

    orientation = (new x=0.0 y=2.5 z=0.0
        y := { @stargate orientation y }
    )
;
```

Listing A.4: gate.dasm

A.2 Wii-fly Script

```
@wiiwidget wiifly = (new
    type = WViewport
    x = 0
    y = 0
    width is { @window width }
    height is { @window height }
    visible = true

    scene = (new
        type = Scene3D
        instances = (new
            height = (new
                type = IHeightmap
                source = (new
                    type = HMImageSource
                    file = (new type=LocalFile
                        filename="wiifly2008/height.png")
                    #Global region material.
                    material = (new
                        texture = (new
                            file = (new type=LocalFile
                                filename="wiifly2008/ground.tga")
                        )
                        diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
                        ambient = (new r=0.7 g=0.7 b=0.7 a=1.0)
                    )
                    materials = (new
                        #Splats
```

```
29                          0 = (new
30                              diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
31                              ambient = (new r=0.7 g=0.7 b=0.7 a=1.0)
32                              textures = (new
33                                  #Splat diffuse
34                                  1 = (new
35                                      file = (new type=LocalFile
36                                      filename="wiifly2008/splat0.tga")
37                                  )
38                                  #Splat alpha
39                                  0 = (new
40                                      file = (new type=LocalFile
41                                      filename=
42                                          "wiifly2008/splat0_alpha.tga")
43                                  )
44                              )
45                          )
46                      )
47                  )
48
49                  scale = (new x=1.0 y=20.0 z=1.0)
50                  position = (new x=0.0 y=-20.0 z=0.0)
51                  patchsize = 16
52                  wireframe is { @keyboard keys tab }
53              )
54
55              skybox = (new
56                  type = IModel
57                  model = (new
58                      type = Model
59                      file = (new type=LocalFile
60                          filename="wiifly2008/skybox.x")
61                      materials = (new
62                          ft = (new
63                              ambient = (new r=0.2 g=0.2 b=0.2 a=1.0)
64                              diffuse = (new r=1.0 g=1.0 b=1.0 a=1.0)
65                          )
66                      )
67                  )
68                  scale = (new x=20.0 y=20.0 z=20.0)
69                  skybox = true
70              )
71              ship = (@ship = (new)
72                  type = IModel
73                  model = (new
74                      type = Model
75                      file = (new
76                          type = LocalFile
77                          filename = "wiifly2008/ship.x"
78                      )
79                  )
80
81                  speed = 30.0
82
83                  position = (new x = 200.0 y = 8.0 z = 200.0
```

```
84                      tx is { @ship speed *
85                              (@math sin (@ship orientation y)) *
86                              (@math cos (@ship orientation x)) }
87                      ty is { @ship speed *
88                              (@math sin (@ship orientation x)) }
89                      tz is { @ship speed *
90                              (@math cos (@ship orientation y)) *
91                              (@math cos (@ship orientation x)) }
92                      x := { .x + (.tx * (@root itime)) }
93                      y := { .y - (.ty * (@root itime)) }
94                      z := { .z + (.tz * (@root itime)) }
95                  )
96                  scale = (new x = 0.3 y = 0.3 z = 0.3)
97
98                  orientation = (new x = 0.0 y = 0.0 z = 0.0
99                      wiix is { @wiimotes 0 axes 0 value }
100                     wiiy is { @wiimotes 0 axes 1 value }
101                     wiiz is { @wiimotes 0 axes 2 value }
102
103
104                     az is { @math atan2 (0.0 - (.wiix)) (.wiiz) }
105                     ax is { @math atan2 (0.0 - (.wiiy))
106                                 (@math sqrt (.wiix *
107                                     (.wiix) + (.wiiz * (.wiiz)))) }
108
109                     dx is {
110
111                         if (.ax < -1.4) -1.4 else {
112                             if (..ax > 1.4) 1.4 else {
113                                 ..ax
114                             }
115                         }
116                     }
117
118                     dz is {
119
120                         if (.az > (.x + 3.14)) {
121                             ..az - 6.28
122                         } else {
123                             if (..az < (..x - 3.14)) {
124                                 ..az + 6.28
125                             } else {
126                                 ..az
127                             }
128                         }
129                     }
130
131                     x := { .x - (.dx + (.x) * (@root itime) * 2.0) }
132                     z := { .z - (.dz + (.z) * (@root itime) * 2.0) }
133                     y := { .y + (@math sin (.dz) *
134                             (@root itime) * 3.0) }
135                 )
136             )
137         )
138     )
```

```
139
140     camera = (@camera = (new)
141         type = Camera3D
142         fov = 90.0
143         near = 1.0
144         far = 500.0
145
146         position = (new x = 8.0 y = -40.0 z = 8.0
147             tx is { @ship position x + (2.0 * (@math cos (0.0 -
148                         (@ship orientation y + 1.57)))) }
149             ty is { @ship position y + 1.0 }
150             tz is { @ship position z + (2.0 * (@math sin (0.0 -
151                         (@ship orientation y + 1.57)))) }
152
153             x := {.x + (.tx - (.x) * 5.0 * (@root itime))}
154             y := {.y + (.ty - (.y) * 5.0 * (@root itime))}
155             z := {.z + (.tz - (.z) * 5.0 * (@root itime))}
156         )
157
158         orientation = (new x = 0.0 y = 0.0 z = 0.0
159             dirx is { @camera position x - (@ship position x) }
160             diry is { @camera position y - (@ship position y) }
161             dirz is { @camera position z - (@ship position z) }
162
163             y is { 0.0 - (@math atan2 (.dirz) (.dirx)) + 1.57}
164             dist is { @math sqrt (.dirx * (.dirx) +
165                         (.dirz * (.dirz))) }
166             x is { 0.0 - (@math atan2 (.diry) (.dist)) }
167         )
168     )
169 );
```

Listing A.5: wiiflygame.dasm

Appendix B

C++ Agents

```
1
2  #ifndef _WGD_SHADER_
3  #define _WGD_SHADER_
4
5  #ifdef WIN32
6  #include <windows.h>
7  #endif
8  #include <GL/gl.h>
9  #include <map>
10 #include <cadence/doste/oid.h>
11 #include <cadence/dstring.h>
12 #include <cadence/agent.h>
13 #include <cadence/file.h>
14 #include <string>
15 #include <wgd/index.h>
16
17
18 namespace wgd {
19
20     class vector3d;
21     class Texture;
22
23 /**
24  * Shader Resource. This contains everything a shader needs to run,
25  * you apply the shader to an object the same way you would apply a texture
26  * with bind() and unbind(). The shader replaces normal textures so if
27  * you use a shader, you dont bind textures directly to the object.<br>
28  * You must specify both vertex shader and fragment shader source, as well as
29  * all the textures that the shader uses. <br>
30  * Shaders that need Binormals and Tangents will automatically get the tangent
31  * if applied to models and primitives, bot you must calculate the binormal
32  * in your vertex shader using: cross(gl_Normal, tangent); <br>
33  * The tangent varying variable must be named "tangent" for it to work.<br>
```

```
34    * If a shader program fails to compile, or will not run on your machine
35    * nothing will happen when you try to bind it.
36    * <br/><br/>
37    *
38    */
39       class RESIMPORT Shader : public cadence::Agent {
40
41           public:
42
43           OBJECT(Agent, Shader);
44
45           Shader();
46           Shader(const cadence::doste::OID &);
47           Shader(cadence::File &vert, cadence::File &frag);
48           Shader(const char* vertfile, const char* fragfile);
49           ~Shader();
50
51           PROPERTY_WF(cadence::File, vert, ix::vert);
52           PROPERTY_RF(cadence::File, vert, ix::vert);
53
54           PROPERTY_WF(cadence::File, frag, ix::frag);
55           PROPERTY_RF(cadence::File, frag, ix::frag);
56
57           PROPERTY_WF(bool, debug, ix::debug);
58           PROPERTY_RF(bool, debug, ix::debug);
59
60           bool make(const char *vert, const char *frag);
61           bool load();
62
63           void bind();
64           void unbind();
65
66           static Shader *current();
67
68           static void current(Shader *sh);
69
70           bool tangents(){ return m_tangents; };
71
72           void setVariable(const char *name, float v1);
73           void setVariable(const char *name, float v1, float v2);
74           void setVariable(const char *name, float v1, float v2, float v3);
75           void setVariable(const char *name, float v1, float v2, float v3, float v4);
76           void setVariable(const char *name, const wgd::vector3d &vec3);
77           void setVariable(const char *name, int v1);
78           void setVariable(const char *name, int size, int* data);
79           void setVariable(const char *name, int size, float* data);
80
81           void enableAttribArray(const char *name);
82           void attribPointer(const char *name, GLint size, GLenum type,
83                GLboolean normalised, GLsizei stride, const void *pointer);
84           void disableAttribArray(const char *name);
85
86           static void enabled(bool);
87           static bool enabled();
88
```

```
89          BEGIN_EVENTS(Agent);
90          EVENT(evt_reload, (*this)("reload"));
91          END_EVENTS;
92
93          private:
94
95          static void initialise();
96          static bool  s_available;
97          static Shader *s_current;
98
99          GLint addVariable(const char *name);
100
101         bool loadShader();
102         char *readFile(cadence::File *);
103         int logInfo(GLuint s, const char *name);
104
105         GLuint m_vertexShader;
106         GLuint m_fragmentShader;
107         GLuint m_program;
108
109         bool m_ready;
110         bool m_loaded;
111
112         bool m_tangents;
113
114         GLint getLocation(const char *name);
115
116         class ShaderVar{
117             public:
118             ShaderVar(int t, GLint loc): type(t), location(loc){};
119             int type; //1=uniform, 2=attribute
120             GLint location;
121         };
122
123         ShaderVar *getVar(const char *name);
124         cadence::doste::OID m_vars;
125
126
127     };
128 };
129
130 #endif
```

Listing B.1: shader.h

Appendix C

Student Questionnaire

Cadence Feedback

Please give a little feedback on the Cadence labs, lecture material and any other experience whilst using it. This is not feedback about Empirical Modelling generally. It will be used anonymously in my PhD thesis to evaluate the use of Cadence for EM and to evaluate the tool itself. It should only take 5 minutes to complete. Please be honest. Thanks.

Which course are you on? *

- ○ MEng
- ○ MSc
- ○ Other:

Gender *

- ○ Male
- ○ Female

Previous Experience

Were you already familiar with Empirical Modelling?

- ○ Yes
- ○ No

Were you already familiar with Cadence?

- ○ Yes
- ○ No

What programming languages did you know before starting the module?

- ☐ Java
- ☐ C/C++
- ☐ Javascript
- ☐ A functional language
- ☐ PHP
- ☐ Ruby
- ☐ Visual Basic
- ☐ C#
- ☐ Other:

Page 2 After page 1 Continue to next page

Cadence Labs

Did you attend the Cadence lab sessions *
Choose yes even if you missed one or two.

- Yes
- No

Page 3 After page 2 Continue to next page

Note: "Go to page" selections will override this navigation. Learn more.

Cadence Labs (Yes)

Were you able to complete most of the labs?

- Yes
- No

Did the "computation as navigation" concept make sense to you?

- Yes
- No

Were you able to understand the dynamic (willbe) definitions?

- Yes
- No

Can you rate the usability of the interface?

	1	2	3	4	5	
Unintuitive	○	○	○	○	○	Very friendly

Rate the simplicity of the DASM notation

	1	2	3	4	5	
Difficult	○	○	○	○	○	Simple and easy

Additional comments on the lab exercises
What was most difficult? What could be improved? What did you think was good about the labs?

Page 4 After page 3 Continue to next page

EM Coursework

Did you use Cadence for the coursework? *

- Yes
- No

Page 5 After page 4 Continue to next page

Note: "Go to page" selections will override this navigation. Learn more.

EM Coursework (Yes)

Was the model successful?
Did it work and achieve some of your objectives?

- Yes
- No

Was it a hybrid model?
Using both Cadence and Eden

- Yes
- No

Did you encounter any technical limitations?
If yes, please give details below

- Yes
- No

Technical Limitations (details)

How was the performance of the tool?

	1	2	3	4	5	
Poor, unusably slow	○	○	○	○	○	Fast, no problems

Was the documentation provided sufficient?
In the form of labs and other provided resources

	1	2	3	4	5	
Inadequate	○	○	○	○	○	Sufficient

What additional documentation would have been most useful?

Page 6 After page 5 Go to page 7 (Final Word)

EM Coursework (No)

Did you attempt to use Cadence for the coursework?

- ○ Yes
- ○ No

What was your reason for not using Cadence?

- ○ Could not get it working
- ○ Notation too difficult to understand
- ○ Missing features
- ○ Not suited to your model
- ○ Lack of confidence in the tool
- ○ Not enough documentation
- ○ Other:

Additional Comments
Please give more detail about any problems encountered, missing features etc

Page 7 After page 6 Continue to next page

Final Word

If you have any scripts and materials left from the labs then it would be much appreciated if you could forward them to myself (nwpope@gmail.com) so that I can analyse them. I am especially interested in any saved histories people have. These materials will not in any way be assessed.

Overall comments on Cadence

Timestamp	Which course are you on?	Gender	Did you attend the Cadence lab sessions	Did you use Cadence for the coursework?	Were you able to understand the dynamic (willbe) definitions?	What programming languages did you know before starting the module?	Were you already familiar with Empirical Modelling?	Were you already familiar with Cadence?	What was your reason for not using Cadence?	Additional Comments	Did you attempt to use Cadence for the coursework?	Was the model successful?	Was it a hybrid model?	Did you encounter any technical limitations?	Technical Limitations (details)	How was the performance of the tool?	Can you rate the usability of the interface?	Were you able to complete most of the labs?	Rate the simplicity of the DASM notation	Additional comments on the lab exercises	Overall comments on Cadence	Was the documentation provided sufficient?	What additional documentation would have been most useful?	Did the "computation as navigation" concept make sense to you?
2/4/2011 17:08:02	MEng	Male	Yes	No	Yes	Java, C/C++, PHP, Visual Basic, C#	Yes	No	Not enough documentation	Although I felt setting up the variable and relationships MAY have been easier in cadence, ultimately I was not going to be able to do anything in cadence I could not already do in eden/donald. Hence, I did not choose to use cadence, although I understood how easy it was to link the two together. I would not have felt confident using cadence alone - not enough documentation.	No						4	Yes	3	The main limitation was a lack of formal technical syntax. Often, the tool was obviously powerful enough to handle, but it was not obvious how to go about achieving what you wanted.				No
2/4/2011 17:30:34	MSc	Male	No	No		Java, Javascript, Visual Basic, C#	Yes	No	Lack of confidence in the tool		Yes													
2/4/2011 18:14:56	MEng	Male	Yes	No	Yes	Java, Javascript, PHP	No	No	Not enough documentation	I think Cadence would have been better but ultimately I did not use it because of the lack of documentation and existing models to fall back on when something didn't work. I was concerned about my mark, and felt there was more support material, and more material in general with tkeden. Had the project been unmarked, I would have used Cadence.	No						4	No	2	They were too long, and I never finished. Maybe I am just slow and the labs are fine though.				Yes
2/4/2011 18:54:56	MSc	Male	Yes	Yes	Yes	Java, C/C++, Javascript, PHP, Visual Basic, C#	No	No				Yes	No	Yes	The interface crashed regularly (segfaults). The model was fairly complicated, taking tenths of a second to update variables, and this was probably the cause of the crashes, as I was not able to track them down to any particular piece of code.	4	4	Yes	3	The concept of navigating a tree should be introduced early; once this was made clear, the language was much easier to use. The interface has many useful features but could use better file handling, and a method of clearing the state of variables to some default (currently only possible by quitting & reopening.	It's a nice language, but needs some improvements, such as functions or arbitrary variable definitions, for example: addone = (new 0 = 1 1 = 2 $misc is ($misc + 1)); now, ".addone 4" is 5, etc. Also brackets are a little arbitrary; I don't think there's any situation where either {} or (could be used, so it should be simplified (I'd suggest replacing {} with ()). Also it would be nice to be able to put brackets on the left (which would be ignored by the language), i.e: a is (((.b) + (.c)) * ((.d) + (.e)); which makes the intention clearer than the current a is {.b + (.c) * (.d + (.e))};	4	Details on working with the WGD would have helped. Trial-and-error, and exploring the variable tree were useful for this, but additional documentation would have helped. Also details about %deep inheritance were not very thorough.	Yes
2/4/2011 18:59:30	ERASMUS	Male	Yes	No	Yes	Java, C/C++, Javascript, PHP	No	No	Not enough documentation	Cadence would be more attractive to students if they could use it as a standalone environment without having to include "ancient" eden	No							3	Yes	2	the labs showed the potential of Cadence maybe it would be helpful to know how easily one can replicate a GUI such as provided by the Stargate model for other purposes	definitely the right approach to aim towards a single tool for EM. so keep on developing!			Yes
2/4/2011 21:57:52	ueos	Male	Yes	Yes	Yes	Java, C/C++, Javascript, A functional language	No	No				Yes	No	Yes	no desired math function, like max, min no special syntax that can used to evaluate insides 'is'/'willbe' definition before assigning the definition to the observables	2	2	Yes	4		i built cadence in my own ubuntu but i found it is very problematic. though i can use it finally, i sometime cannot load particular script, and the WGD lib cannot perform well in my machine (the screen did not display correctly, and sometime force quit). in many cases i also faced segmentation fault. in my development process, i kept thinking to change to Eden environment because it gives us much more support, not only documentation but also functional support. there is not much tutorial on how to use agents, or handlers or some more adv use of the tool like building up our own module/lib/function anyway, it was a very nice experience. and i do suggest using Cadence. looking forward to the improvement of Cadence. Thanks	2		Yes
2/5/2011 13:04:37	MSc	Male	Yes	No	Yes	C/C++	Yes	No	Lack of confidence in the tool		No							3	Yes	3					No
2/5/2011 23:30:02	MSc	Male	Yes	No	Yes	Java, C/C++, Javascript, PHP	No	No	Not enough documentation		No							4	Yes	3					Yes
2/6/2011 14:14:02	MSc	Male	Yes	No	Yes	Java, C/C++, Javascript	No	No	Not enough documentation		No							3	Yes	3					No
5/8/2011 23:03:43	MSc	Male	Yes	No	Yes	Java, C/C++, Javascript, Visual Basic, C#	No	No	Could not get it working	I tried to used, but some initially behaviours did not worked as I expected. However it see as a very good approach to model interactive environments.	Yes							3	Yes	2	Richer development, debugging tools,	I tried to used, but some initially behaviours did not worked as I expected. However it see as a very good approach to model interactive environments.			No

Bibliography

M. Abi-Antoun and T. Selitsky. Interactive Refinement of Runtime Structure. Flexitools [at] SPLASH 2010 workshop http://www.ics.uci.edu/~nlopezgi/flexitools/papers/abiantoun_flexitools_splash2010.pdf [Accessed 29/7/2011], 2010.

L. W. Beng. Teaching about water supply. WEB-EM 2 http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/02/water.pdf [Accessed 2/6/2011], 2006.

M. Beynon. Cabinet digit visual pun. http://empublic.dcs.warwick.ac.uk/projects/cabinetdigitBeynon1990/ [Accessed 23/5/2011], 1990.

M. Beynon. Lines. http://empublic.dcs.warwick.ac.uk/projects/linesBeynon1991/ [Accessed 2/6/2011], 1991.

M. Beynon. Concurrent Systems Modelling: Agentification, Artefacts, Animation. Lecture Notes http://empublic.dcs.warwick.ac.uk/projects/emfcsBeynon1997/MSc2001/MSc92-9/TUESDAY/NOTES/tue_lecture3.htm [Accessed 23/5/2011], 1997a.

M. Beynon. The LSD Notation for Agent Specification. Lecture Notes http://empublic.dcs.warwick.ac.uk/projects/emfcsBeynon1997/MSc2001/MSc92-9/TUESDAY/NOTES/tue_lecture4.htm [Accessed 4/6/2011], 1997b.

M. Beynon. Artefacts in Visualisation and Concurrent Systems Modelling. Lecture Notes http://empublic.dcs.warwick.ac.uk/projects/emfcsBeynon1997/

MSc2001/MSc92-9/TUESDAY/NOTES/tue_lecture5a.htm [Accessed 4/6/2011], 1997c.

M. Beynon. Monotone Boolean Functions in 4 variables. http://empublic.dcs.warwick.ac.uk/projects/mbf4Beynon2003/ [Accessed 8/2/2011], 2003.

M. Beynon. Erlkoenig model. http://www.dcs.warwick.ac.uk/~wmb/webeden/Erlkoenig.html [Accessed 2/8/2011], 2006a.

M. Beynon. Mathematics and Music - Models and Morals. In *Bridges London: Mathematical Connections in Art, Music, and Science*, pages 437–444. Tarquin Books, 2006b.

M. Beynon. *Modelling with experience: construal and construction for software*, chapter 4. Ways of Thinking. Springer-Verlag, 2011. to appear.

M. Beynon and R. Cartwright. Empirical modelling principles for cognitive artefacts. *IEE Seminar Digests*, 1995(231):8–8, 1995.

M. Beynon and Z. E. Chan. A conception of computing technology better suited to distributed participatory design. NordiCHI Workshop on Distributed Participatory Design, 2006.

M. Beynon and N. Pope. Cadence and the Empirical Modelling conceptual framework: a new perspective on modelling state-as-experienced. Technical Report CS-RR-447, Department of Computer Science, University of Warwick, 2011.

M. Beynon and S. Russ. Redressing the past: liberating computing as an experimental science. Technical Report CS-RR-421, Department of Computer Science, University of Warwick, 2006.

M. Beynon, R. Boyatt, and Z. E. Chan. Intuition in Software Development Revisited. In *20th Annual Psychology of Programming Interest Group Conference*, 2008.

W. M. Beynon. A Definition of the ARCA Notation. Technical Report CS-RR-54, Department of Computer Science, University of Warwick, 1983.

W. M. Beynon. Definitive notations for interaction. In *HCI 85*, pages 23–34. Cambridge University Press, 1985.

W. M. Beynon. ARCA- A Notation for Displaying and Manipulating Combinatorial Diagrams. Technical Report CS-RR-78, Department of Computer Science, University of Warwick, 1986a.

W. M. Beynon. The LSD notation for communicating systems. Technical Report CS-RR-87, Department of Computer Science, University of Warwick, 1986b.

W. M. Beynon. Parallelism in a definitive programming framework. In *Proceedings of the International Parallel Computing Conference 89*, pages 425–430, Leiden, August 1989.

W. M. Beynon. Agent-oriented Modelling and the Explanation of Behaviour. In *International Workshop: Shape Modelling Parallelism, Interactivity and Applications*, pages 54–63, University of Aizu, Japan, 1994.

W. M. Beynon. Radical Empiricism, Empirical Modelling an the nature of knowing. In *Cognitive Technologies and the Pragmatics of Cognition*, pages 155–184. 2007.

W. M. Beynon and A. Harfield. Constructionism through Construal by Computer. In *Constructionism 2010*, Paris, France, 2010.

W. M. Beynon and S. Russ. The Interpretation of States: a New Foundation for Computation? Technical report, University of Warwick, Coventry, UK, UK, 1992.

W. M. Beynon and S. B. Russ. Empirical Modelling of Requirements. Technical Report CS-RR-277, University of Warwick, 1995.

W. M. Beynon, J. Rungrattanaubol, and J. Sinclair. Formal Specification from an Observation-Oriented Perspective. *Universal Computer Science*, 6(4):407–421, 2000a.

W. M. Beynon, A. Ward, S. Maad, A. Wong, S. Rasmequan, and S. Russ. The Temposcope: a Computer Instrument for the Idealist Timetabler. In *Proceedings of the 3rd international conference on the practice and Theory of Automated Timetabling*, pages 153–175, Konstanz, Germany, 2000b.

W. M. Beynon, R. Boyatt, and S. Russ. Rethinking Programming. In *Proceedings IEEE Third International Conference on Information Technology: New Generations (ITNG 2006)*, pages 149–154, 2006a.

W. M. Beynon, S. Russ, and W. McCarty. Human Computing: Modelling with Meaning. *Literary and Linguistic Computing*, 21(2):141–157, 2006b.

M. Black. *Models and Metaphors*, chapter 13. Cornell University Press, 1962.

P. Buneman. Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '97, pages 117–121, New York, NY, USA, 1997. ACM.

M. M. Burke. Prototype-based languages. *J. Comput. Small Coll.*, 21:215–216, December 2005.

M. Burnett. What Is End-User Software Engineering and Why Does It Matter? In V. Pipek, M. Rosson, B. de Ruyter, and V. Wulf, editors, *End-User Development*, volume 5435 of *Lecture Notes in Computer Science*, pages 15–28. Springer Berlin / Heidelberg, 2009.

M. Burnett, J. Atwood, R. W. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.

G. Calvary, J. Coutaz, O. Dassi, L. Balme, and A. Demeure. Towards a New Generation of Widgets for Supporting Software Plasticity: The Comet. In *Engineering for Human-Computer Interaction*, pages 306–324, 2004.

C. Care. The Analogue Computer as a Scientific Instrument. Technical Report CS-RR-420, University of Warwick, 2006.

R. Cartwright. Digital Watch and Chess Clocks. http://empublic.dcs.warwick.ac.uk/projects/digitalwatchCartwright1995/ [Accessed 24/5/2011], 1995.

R. Cartwright. *Geometric Aspects of Empirical Modelling: Issues in Design and Implementation*. PhD thesis, University of Warwick, 1999.

R. Cartwright, V. Adzhiev, A. Pasko, Y. Goto, and T. L. Kunii. Web-based shape modeling with hyperfun. *IEEE Computer Graphics and Applications*, 25:60–69, 2005.

Z. E. Chan. *Towards efficacious groupware development: an Empirical Modelling approach*. PhD thesis, University of Warwick, 2009.

S. Chiba and R. Ishikawa. Aspect-Oriented Programming Beyond Dependency Injection. In *Proceedings of ECOOP 2005*, pages 121–143, 2005.

S. Cox. WPF / WF What Is A Dependency Property? Blog post at: http://techpunch.wordpress.com/2008/09/25/wpf-wf-what-is-a-dependency-property/ [Accessed 8/8/2011], September 2008.

W. Dangerfield. Investigating the use of EM to model SCUBA diving. CS405 Coursework Paper, Department of Computer Science, University of Warwick, 2011.

M. Dertouzos. Creating The People's Computer. *MIT Technology Review*, 1997.

M. Desmond, H. Ossher, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, and S. Krasikov. Towards smart office tools. Flexitools [at] SPLASH 2010 workshop http://www.ics.uci.edu/~nlopezgi/flexitools/papers/desmond_flexitools_splash2010.pdf [Accessed 16/7/2011], 2010.

T. Dingsyr, T. Dyb, and N. B. Moe, editors. *Agile Software Development: Current Research and Future Directions*. Springer, 2010.

Y. Dittrich, O. Lindeberg, and L. Lundberg. End-User Development as Adaptive Maintenance. In *End-User Development*, volume 9 of *Human Computer Interaction Series*, pages 295–313. Springer, 2006.

R. W. Djang and M. M. Burnett. Similarity inheritance: A new model of inheritance for spreadsheet vpls. In *IEEE Symposium on Visual Languages*, pages 134–141, 1998.

R. W. Djang, M. M. Burnett, R. D. Chen, and D. Chen. Static Type Inference for a First-Order Declarative Visual Programming Language with Inheritance. *Journal of Visual Languages and Computing*, 11:191–235, 2000.

J. R. Douglass. Language of Languages for Flexible Development. Flexitools [at] SPLASH 2010 workshop `http://www.ics.uci.edu/~nlopezgi/flexitools/papers/douglass_flexitools_splash2010.pdf` [Accessed 16/7/2011], 2010.

J. Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 505–518, New York, NY, USA, 2005. ACM.

J. Edwards. Coherent reaction. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 925–932, New York, NY, USA, 2009. ACM.

J. Edwards. Alarming development. `http://alarmingdevelopment.org/` [Accessed 20/5/2011], 2010.

EM Projects. Empirical Modelling. `http://www2.warwick.ac.uk/fac/sci/dcs/research/em/projects/` [Accessed 17/05/11].

EM Website. Empirical Modelling. `http://www.dcs.warwick.ac.uk/modelling` [Accessed 13/08/10].

T. Erl. *Service-oriented architecture : concepts, technology, and design*. Prentice Hall, 2005.

D. Evans. Modelling a Procedural Calculator Using Definitive Methods. CS405 Coursework Paper, Department of Computer Science, University of Warwick, 2011.

C. N. Fischer and M. Beynon. Empirical Modelling of Products. In *Proceedings of the International Conference on Simulation and Multimedia in Engineering Education*, pages 27–32, Phoenix, Arizona, 2001.

G. Fischer. Meta-DesignDesign for Designers. In *3rd International Conference on Designing Interactive Systems (DIS 2000*, pages 396–405. ACM Press, 2000.

G. Fischer. End-User Development and Meta-Design: Foundations for Cultures of Participation. In *Proceedings of the 2nd International Symposium on End-User Development*, pages 3–14, Berlin, 2009. Springer-Verlag.

G. Fischer and E. Giaccardi. Meta-design: A Framework for the Future of End-User Development. In *End-User Development*, volume 9 of *Human Computer Interaction Series*, pages 427–457. Springer, 2006.

D. Gooding. *Experiment and the Making of Meaning: Human Agency in Scientific Observation and Experiment*. Springer, 1990.

D. Gooding. Experiment as an instrument of innovation: Experience and embodied thought. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind*, CT '01, pages 130–140, Coventry, UK, 2001. Springer-Verlag.

T. Green and M. Petre. Usability analysis of visual programming environments: A Cognative Dimensions framework. *Visual Languages and Computing*, 7(2):131–174, 1996.

S. Hammond. Neural Networks and Notations. WEB-EM 2 `http://www2.warwick.`

ac.uk/fac/sci/dcs/research/em/publications/web-em/02/neural.pdf [Accessed 2/6/2011], 2006.

D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

A. Harfield. Agent-oriented Parser. http://empublic.dcs.warwick.ac.uk/projects/agentparserHarfield2003/ [Accessed 20/7/2011], 2003.

A. Harfield. Guide to the Agent-Oriented Parser. http://www2.warwick.ac.uk/fac/sci/dcs/research/em/notations/aop/guide [Accessed 20/7/2011], 2006a.

A. Harfield. Dependency Viewer. http://empublic.dcs.warwick.ac.uk/projects/dmtHarfield2006/ [Accessed 20/7/2011], 2006b.

A. Harfield. Presentation Environment. http://empublic.dcs.warwick.ac.uk/projects/empeHarfield2007/ [Accessed 21/6/2011], 2007a.

A. Harfield. Sudoku colour. http://empublic.dcs.warwick.ac.uk/projects/sudokucolourHarfield2007/ [Accessed 8/6/2011], 2007b.

A. Harfield. *Empirical Modelling as a new paradigm for educational technology*. PhD thesis, University of Warwick, 2008.

A. Harfield. Dependency in Action. Presentation [Access Restricted] http://www2.warwick.ac.uk/fac/sci/dcs/research/em/teaching/cs405-0910/dependencyinaction.ppt [Accessed 8/8/2011], 2009.

A. Harfield, M. Beynon, and R. Myers. Web Eden and Moodle: an Empirical Modelling approach to web-based education. In *Proceedings of the Eighth IASTED International conference on Web-Based Education*, pages 272–278, March 2009.

U. Hölzle and D. Ungar. A third-generation self implementation: reconciling responsiveness with performance. In *Proceedings of the ninth annual conference on Object-*

oriented programming systems, language, and applications, OOPSLA '94, pages 229–243, New York, NY, USA, 1994. ACM.

G. C. Hunt and J. R. Larus. Singularity Design Motivation. Technical Report MSR-TR-2004-105, Microsoft Research, 2004.

M. Jackson. Problem frames and software engineering. *Inf. Softw. Technol.*, 47(14): 903–912, November 2005.

T. Jacobs. Belief in Balance of Nature Hard to Shake. Miller-McCune `http://www.miller-mccune.com/science-environment/belief-in-balance-of-nature-hard-to-shake-4785/` [Accessed 4/6/2011, December 2007.

W. James. *Essays in Radical Empiricism*. Longmans, Green and Co., New York, 1912/1996.

C. Kalogirou. How to do good bloom for hdr rendering. Blog `http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/` [Accessed 15/6/11], 2006.

A. Kay. Computer Software. *Scientific American*, 251(3):52–59, September 1984.

C. Keen. Third year project oral timetabling. `http://empublic.dcs.warwick.ac.uk/projects/projecttimetableKeen2000/` [Accessed 26/7/2011], 2000.

D. Keer. Modelling Navigation and Landmarking in Ants. `http://empublic.dcs.warwick.ac.uk/projects/antnavigationKeer2010/` [Accessed 8/6/2011], 2010.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, Finland, 1997. Springer-Verlag.

D. King. *Parting Software and Program Design*. PhD thesis, University of York, 2005.

K. King. Uncovering Empirical Modelling. Master's thesis, University of Warwick, 2004.

A. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The State of the Art in End-User Software Engineering. *ACM Computing Surveys*, 43(3), 2011.

N. Lee. ELS – An Eden Based Digital Logic Simulator. WEB-EM 3 `http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/03/logicsim.pdf` [Accessed 2/6/2011], 2007.

M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

M. Lehman. Feedback in the software evolution process. *Information and software technology*, 38:681–686, 1996.

H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: http://doi.acm.org/10.1145/28697.28718. URL `http://doi.acm.org/10.1145/28697.28718`.

H. Lieberman, editor. *Your wish is my command: programming by example*. Morgan Kaufmann, 2001.

H. Lieberman, F. Paterno, and M. Klann. End-User Development: An Emerging Paradigm. In *End-User Development*, volume 9 of *Human Computer Interaction Series*, pages 1–8. Springer, 2006.

J. Maloney. Morphic: The Self User Interface Framework. ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf [Accessed 14/5/11], 2000.

M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

M. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical report, Combex Inc, 2003.

R. Milner. Is Computing an Experimental Science? Technical Report ECS-LFCS-86-1, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

S. Morris. Software Plasticity with Aspect-Oriented Programming. http://www.informit.com/articles/article.aspx?p=413094&seqNum=2 [Accessed 14/4/11], September 2005.

B. Nardi. *A Small Matter of Programming*. MIT Press, 1993.

P. Ness. *Creative Software Development: An Empirical Modelling Framework*. PhD thesis, University of Warwick, 1997.

D. Norman. *The Design of Everyday Things*. Basic Books, New York, 1998.

J. F. Pane and B. A. Myers. More Natural Programming Languages and Environments. In *End-User Development*, volume 9 of *Human Computer Interaction Series*, pages 31–50. Springer, 2006.

A. J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9):7–13, September 1982.

R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. Technical report, Bell Laboratories, 1995.

N. Pope. A definitive notation for behaviour in Empirical Modelling? WEB-EM 3 http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/03/doste.pdf [Accessed 23/5/2011], 2007.

N. Pope. Cadence source code. GitHub Repository `https://github.com/knicos/Cadence` [Accessed 13/6/2011], 2011.

N. Pope and M. Beynon. Cadence lab exercises. `http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/cadence/labs/` [Accessed 8/2/2011], 2010a.

N. Pope and M. Beynon. Empirical Modelling as an unconventional approach to software development. FlexiTools Workshop, October 2010b.

A. Repenning and A. Ioannidou. What Makes End-User Development Tick? 13 Design Guidelines. In *End-User Development*, volume 9 of *Human Computer Interaction Series*, pages 51–85. Springer, 2006.

A. Repenning, A. Ioannidou, and J. Zola. AgentSheets: End-User Programmable Simulations. *Journal of Artificial Societies and Social Simulation*, 3(3), 2000.

M. Resnick, J. Maloney, A. Monroy-Hernndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, November 2009.

C. Roe. *Computers for Learning: An Empirical Modelling Perspective*. PhD thesis, University of Warwick, 2003.

C. Roe, M. Beynon, and C. N. Fischer. Empirical Modelling for the Conceptual Design and Use of Engineering Products. In *Proceedings of the International Conference on Simulation and Multimedia in Engineering Education*, pages 20–26, Phoenix, Arizona, 2001.

G. Rogers. *The Nature of Engineering: A Philosophy of Technology*. Palgrave Macmillan, 1983.

A. Rosenfeld. *An introduction to algebraic structures*. Holden-Day, 1968.

J. Rungrattanaubol. *A treatise on Modelling with definitive scripts*. PhD thesis, University of Warwick, 2002.

M. Sendn, J. Lors, T. Granollers, and F. Perdrix. Implicit Plasticity Framework: a Client-Side Reusable Software Architecture for Context-Awareness, 2005.

M. Slade. Definitive Parallel Programming. Master's thesis, University of Warwick, 1990.

B. C. Smith. Two lessons of logic. *Computational Intelligence*, 3(1):214–218, 1987.

B. C. Smith. *On the Origin of Objects*. The MIT Press, 1996.

R. B. Smith, A. B. Smith, and D. Ungar. Programming as an Experience: The Inspiration for Self. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 303–330, arhus, Denmark, 1995. Springer-Verlag.

M. Spahn, C. Dorner, and V. Wulf. End User Development: Approaches Towards a Flexible Software Design. In *Proceedings of ECIS 2008*. `http://aisel.aisnet.org/ecis2008/189`, 2008.

F. Steinle. Entering New Fields: Exploratory Uses of Experimentation. In *Proceedings of the Biennial Meetings of the Philosophy of Science Association*, volume 64. The University of Chicago Press, 1997.

P.-H. Sun. *Distributed Empirical Modelling and its Applications to Software System Development*. PhD thesis, University of Warwick, 1999.

W. Swartout and R. Balzer. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, 1982.

S. L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2):127–139, 1990.

D. Truex, R. Baskerville, and J. Travis. Amethodical systems development: the deferred meaning of systems development methods. *Accounting Management and Information Technologies*, 10:53–79, 2000.

D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

UNK. Agent Based Bridges. WEB-EM 1 `http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/01/agentbridges.pdf` [Accessed 2/6/2011], 2005.

W. Wadge and E. Ashcroft. *LUCID: The Data Flow Programming Language*. Academic Press Inc., 1985.

A. Ward. *Interaction with Meaningful State: Implementing Dependency on Digital Computers*. PhD thesis, University of Warwick, 2004.

E. W. Weisstein. Dynamical Systems. MathWorld, A Wolfram Web Resource `http://mathworld.wolfram.com/DynamicalSystem.html` [Accessed 8/6/2011].

Wikia. Stargate. Wikia `http://stargate.wikia.com/wiki/Stargate` [Accesssed 15/6/11], 2011.

R. J. Wilson. Simulating the Kinesin Walk: A Small Step towards Understanding Dementia. In *Computer Modeling and Simulation, 2008. EMS '08. Second UKSIM European Symposium on*, pages 226 –231, sept. 2008.

A. Wong. WING. `http://empublic.dcs.warwick.ac.uk/projects/wingWong1998/` [Accessed 5/6/2011], 1998.

A. Wong. EME. `http://empublic.dcs.warwick.ac.uk/projects/emeWong2001/` [Accessed 5/6/2011], 2001.

A. Wong. *Before and Beyond Systems: An Empirical Modelling Approach*. PhD thesis, University of Warwick, 2003.

E. Yung. Room. `http://empublic.dcs.warwick.ac.uk/projects/roomYung1989/` [Accessed 20/7/2011], 1989.

E. Yung. EDEN: An Engine for Definitive Notations. Master's thesis, University of Warwick, 1990.

S. Yung. *Definitive Programming: a Paradigm for Exploratory Programming*. PhD thesis, University of Warwick, 1993.